

软件高级编程实例精解丛书

C++ Builder 5 高级编程实例精解

刘滨 编著　康创 策划



国防工业出版社

Digitized by the Internet Archive
in 2024

https://archive.org/details/isbn_9787118024906

软件高级编程实例精解丛书



C++ Builder 5 高级编程实例精解

刘 滨 编著

康 创 策划

郑州防空兵学院研究生队2002级

周海军

杨宁

2004.12.20

国防工业出版社

·北京·

内容简介

本书通过一系列规模较大并具有现实意义的实例，深入系统地介绍了使用Borland C++ Builder 5 开发 Win32 应用程序的各项关键技术。C++ Builder是新一代基于C++语言的可视化开发工具，在高性能的执行效率与底层控制和快速可视化开发两方面均表现出色。

全书共分17章，第1章是对C++ Builder基础性问题的深入讨论和应用C++ Builder 5新增技术编程，随后的每个章节均通过一个精彩实例深入讲述C++ Builder某个方面的编程技术及技巧。这些实例包括基于CGI和ISAPI的网页留言簿系统/图形计数器/聊天室等服务器端程序、利用钩子函数实现的截图工具、基于DirectX技术的Block游戏程序等，内容涵盖C++ Builder编程的方方面面，并重点介绍了计算机网络编程技术、多媒体编程和COM、DirectX等新技术编程。

本书是为那些对C++ Builder编程有所了解并想学习更高技术和编程技巧的读者编写的，并可作为软件开发人员的参考资料。如果您是C++ Builder的初学者，本书可能并不适合您。

图书在版编目(CIP)数据

C++ Builder 5 高级编程实例精解/刘滨编著.—北京:国防工业出版社,2001.7
(软件高级编程实例精解丛书)
ISBN 7-118-02490-2

Ⅰ.C… Ⅱ.刘… Ⅲ.C语言-程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2001)第 06391 号

国防工業出版社 出版发行
(北京市海淀区紫竹院南路 23 号)
(邮政编码 100044)
北京奥隆印刷厂印刷
新华书店经售
*
开本 787×1092 1/16 印张 38½ 885 千字
2001 年 7 月第 1 版 2001 年 7 月北京第 1 次印刷
印数:1-3000 册 定价:55.00 元

(本书如有印装错误，我社负责调换)

总 序

该丛书从策划到交稿，比预定时间拖后了许多。

正如开发项目的时间往往不可控制。

该丛书就如做项目；不是一个项目，而是一个又一个的项目。

在做项目的过程中，不是因为困难，而是发现可以更上一层楼，于是一再完善，也因此一再延后。

丛书一开始便本着“通往神殿之路”这一开发最高境界的宗旨铺开，力求摆脱市场上触目可及的面对所有层次读者的 “软件开发丛书”的窠臼；即脱开初级用户，紧密针对高级读者，或者针对想深究该开发软件者。

成书之后，虽然我们不能夸下海口，但却可以绝对肯定地告诉你：物超所值！

尤其是那些程序，凝结了我们多年来的开发经验，读者定能从中获益匪浅。

当然，毕竟时间有限，版本升级又迫在眉睫，我们只能放弃再完善的想法了。

但仍然希望读者们能扶桌感慨——

市面上，像这样的好书真不多！

在这里，我特别向以下合作者表示最真心的感谢：

刘国栋，该丛书质量编辑，全面审查书稿的内容与形式，不辞辛苦毫无怨言。

刘滨，作者，为了写好该书，做出了很大牺牲，但仍然坚持顶住。

曾杰、王海东等，作者，在一再的大改稿要求面前，仍然耐心接受并付诸行动。

同时还有许多提出一系列建议者，一并感谢。

当然，丛书肯定有许多不尽如人意或错误纰漏之处，敬请指教；另外，如果你对该丛书有什么建议或意见，请与我们联系，联系方式如下：

adrain@263.net

我们将继续奉献这些“物超所值”的高级实例精解丛书。

曾满平

前　言

Borland C++ Builder 是 C++语言发展的一个划时代的里程碑，在短时间内已发展到第 5 代。Borland C++ Builder 5 结合了 Borland 公司的两个顶级产品——Delphi 5 和 Borland C++ 5.5 的优点，开创性地将可视化开发环境、可视双向开发工具和广受好评的可视组件库加入 C++语言中，造就出在高效的底层控制与快速可视化开发两方面都表现出色的新一代 C++开发工具。

融入精彩纷呈的实例

优秀的实例程序在编程语言学习中的作用是极其巨大的。单纯的讲解或简单的演示性示例往往不够深刻，面临实际开发时还是会遇到这样那样的问题难以解决。基于此点，本书将通过一系列精彩纷呈、令人印象深刻的典型实例，讲述使用 Borland C++ Builder 5 开发 Windows 应用程序的各项高级技术。具体来说，本书每个专题都将实现一个较大规模并具有实际意义的主要范例程序，同时还包括一些小而精练的实例，说明主要实例无法说明的关键性问题。融入这些精彩纷呈的实例程序，将使你能够透彻掌握 C++ Builder 编程技术及技巧，快速成为高级开发者。

体验高级开发的乐趣

C++ Builder 是 Windows 可视化编程技术和高效灵活的 C++语言的完美结合。C++ Builder 5 正是建立在百分之百纯 C++语言和 Delphi 中广受好评的 VCL 的基础上，提供了一种令人惊奇而且无所不包的软件开发工具。无论是多媒体、数据库、客户机/服务器、Intranet 或 Internet 解决方案，还是更快的程序建立速度、更高效的执行代码、更完善的开发环境，在 C++ Builder 中都有出色表现。使用 C++ Builder 开发应用程序无疑是非常适宜的。

本书通过一系列出色的实例程序深入浅出地讲解 Borland C++ Builder 5 编程的方方面面。书中所有主要实例均注意范例程序的实用性、典型性和趣味性。这些主要实例程序具有很强的实际应用背景，并且大部分实例已经是一个具备强大功能的实用软件。本书的各个章节将详细分析这些实例程序的开发，并围绕实例讲述关键知识点和编程技术。通过精心设计的精彩例程，你将轻松地学习并精通 C++ Builder 编程，并体验高级开发的乐趣。

记住，你是未来的高级开发者

本书主要是为那些对 C++ Builder 编程有所了解并想学习高级技术和技巧的读者所编写，同时也是很有价值的软件开发人员的参考资料。本书不是 C++ Builder 5 的入门参考书，如果你是 C++ Builder 的初学者，本书可能并不适合于你。

最好地发挥该书的作用

本书共分 17 章，深入透彻地介绍 C++ Builder 5 在各个领域的编程技术。

第 1 章从开发工具特色及编程理念方面对 Borland C++ Builder 进行系统的分析，使你对 C++ Builder 这一优秀开发工具有更加深刻的认识。同时深入讨论 Visual Component Library 的高级话题和 C++ Builder 常用抽象数据类，并介绍如何使用 C++ Builder 5 引入的新增技术编程。

第 2 章将编程实现一个比 Windows 95/98 的写字板更好的字处理应用程序——BCB 书写器，通过实例程序，讲述关于文本编辑处理方面的编程问题以及多文档界面（MDI）技术。

第 3 章讲述 C++ Builder 文件相关编程，并创建和 Windows 95 的资源管理器相同的 BCB 文件管理器程序。同时这一章还将讲述 C++ Builder 文件流、与文件相关的重要 API 函数以及 Win32 外壳 API 等重要编程技术。

第 4 章将通过对一个精彩范例——BCBSee32 图像浏览和转换工具的分析，讲述 C++ Builder 多格式图像显示、图像特性控制、格式转换等编程技术。

第 5 章将通过一个比 Windows 95/98 画图工具（MSPaint）更强大的范例程序——BCB 画板，讲述 C++ Builder 中图像绘制、图像编辑和图像数字处理编程技术。

第 6 章使用 C++ Builder 完成一个 Windows 屏幕保护程序，重点讲解 Windows 动画技术和图形技巧显示的内容，包括 Windows 消息的拦截和处理、利用注册表、应用程序参数传递等 Windows 编程技术。

第 7 章将实现一个漂亮的程序——BCBPlayer 多媒体播放器，并阐述如何利用 C++ Builder 环境中嵌入的多媒体组件和 Windows 多媒体 API 来实现这一切。同时这一章还将介绍应用 MCI 和底层多媒体 API 编程。

第 8 章是 VCL 编程技术的综合应用，届时将开发一个真正完整的 Windows 游戏程序——俄罗斯方块。该范例程序支持游戏手柄操作，并且具备 Windows 游戏所需的所有要素——背景音乐及音效、英雄榜系统、帮助系统、可更换背景图案以及良好的操作方式。

第 9 章讲述 C++ Builder 中进程和多线程编程技术，多线式文件处理工具是这一章将要开发的实用工具，它包括快速文件查找和快速文件拷贝两个模块，完全利用了 Win32 的多线程技术。

第 10 章讲述 Internet 协议以及 Internet 客户方应用程序编程技术。所完成的主要实例是一个漂亮并功能齐全的 HTML 浏览器——BCB WebBrowser，同时还将分析邮件协议客户端程序和 FTP 客户端程序的实现。

第 11 章介绍 Web 服务器端编程的各种技术，包括使用 C++ Builder 编写 CGI 和 ISAPI 程序。主要实例是一个完善的网页留言簿系统。它通过动态生成 HTML 页面，支持留言簿数据库的建立、浏览和管理。

第 12 章将创建可连网对战的游戏程序——网络五子棋。通过这个程序的实现，讲述 C++ Builder 中应用 WinSock 组件——TClientSocket 组件和 TServerSocket 组件的编程技术。同时涉及 WinSock 编程的一些高级问题，以及 C++ Builder 的 WinSock 组件替代技术——更为强劲的 PowerSock 组件。

第 13 章讲述 VCL 组件的开发。实例是一个功能强大的图像时钟组件 TCoolClock。该组件实现了一个大小、样式和颜色可调的实时时钟，同时本章还包括其他一些有价值的组件范

例。

第 14 章完成一个功能强大的“系统环境监视程序”，实现包括特殊风格的界面、查看当前顶级窗口、动态汉化窗口菜单、查看驱动器信息、查看内存资源信息、查看/调整显示状态、查看/设置系统环境变量、查看正在运行的程序信息、杀除进程等多项功能，通过对该范例程序的分析，你将了解调用 Windows API 的各种方式以及几个重要的 Windows API 技巧和许多有用的 API 函数。

第 15 章完成名为“BCB 抓图大师”的实用截图工具。“BCB 抓图大师”能够实现多种截图方式和多图像格式的屏幕截图，同时具备截取图像的浏览功能。通过该程序，系统讲述了 Windows 钩子函数的使用以及进程间利用映像文件共享数据的技术。

第 16 章讲述组件对象模型（COM）技术，并提供多个应用实例以演示 COM 技术的各个方面。包括编写可以供多种编程语言使用的代码（COM 对象），自动化 Word 及 Excel 应用程序，利用 COM 接口实现更为强大的 BCB Web Browser 程序的第三版本，以及创建 ActiveX 控件 CoolClockX。

第 17 章深入讲述 DirectX 编程技术，包括 DirectDraw、Direct3D、DirectSound 编程问题。这一章将创建一个真正基于 DirectX 技术的游戏程序——Block 游戏，这是一个声光效果俱佳并很有意思的接打球游戏。

附带光盘价值多多

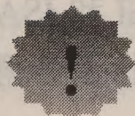
❖ 目录结构说明

本书附带光盘包含了书中 17 章近一百个实例程序的源代码和相关资源。光盘有两个主目录，其中 Install 目录下是该项目的安装程序，可通过安装程序选择安装各章节的实例程序源码，同时该安装程序还包含 C++ Builder 5 的主要运行库文件和包文件，可以保证绝大多数程序脱离 C++ Builder 5 环境运行。Source 目录下是这些源程序和已经编译好的可执行程序的拷贝版，方便您随时查询和运行。

❖ 查询源程序

Source 目录下的源程序按章节组织。各章节主要实例程序存在的目录以该项目名称命名，其他实例程序目录一般以“EX 项目序号”方式命名。结构清晰，便于查询。

使用本书的几个约定



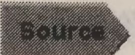
该图标标示出提醒读者应该注意的文字。



该图标标示出对读者具有一定提示和启发性的文字。



该图标标示出对书中某些内容作补充说明的文字。



该图标标示出书中出现的源代码。

目　录

第 1 章　C++ Builder 5 深入剖析……1
1.1　C++ Builder——伟大的开发工具……3
1.2　VCL 高级话题……6
1.2.1　深入 TObject 类……6
1.2.2　深入 TApplication 类……10
1.2.3　深入 TForm 类……13
1.2.4　TMetaClass 和类引用……16
1.3　抽象数据类型……16
1.3.1　链表（TList）……17
1.3.2　字符串（AnsiString）……22
1.3.3　集合（Set）……23
1.3.4　动态数组（DynamicArray）……25
1.3.5　流（TStream）……26
1.4　使用 C++ Builder 5 的 VCL 增强……27
1.4.1　C++ Builder 5 的新特点……27
1.4.2　使用 TActionList 组件和 TMonthCalender 组件……30
1.4.3　ADOExpress 组件编程……33
1.4.4　框架（Frame）技术编程……36
第 2 章　功能齐全的多文档书写器——高级文本处理……39
2.1　文本编辑组件的高级用法……41
2.1.1　TEdit 组件和 TMaskEdit 组件……41
2.1.2　TMemo 组件和 TRichEdit 组件……43
2.2　多文档界面（MDI）和多页面界面（MPI）技术……45
2.2.1　多文档接口与 MDI 应用程序……45
2.2.2　多页面界面 MPI……46
2.3　实例创建分析……48
2.4　创建 MDI 的编辑环境……49
2.4.1　主窗体和子窗体界面……50
2.4.2　实现可停驻（Docking）工具条……51
2.4.3　菜单融合处理和窗体布局控制……53
2.5　基本文本编辑功能的实现……54
2.5.1　文档的打开、存盘、关闭和打印……54
2.5.2　剪贴板编辑功能……58

2.6 字体格式控制、查找与替换 …… 59
2.6.1 字体和段落格式控制 …… 60
2.6.2 查找与替换 …… 64
2.7 实现高级功能 …… 67
2.7.1 历史文件列表菜单 …… 68
2.7.2 当前光标所在行、列数的报告 …… 70
2.7.3 实现 MDI 父窗体的背景贴图 …… 71
第 3 章 完整的文件管理器——文件操作和文件流 …… 74
3.1 C++ Builder 的文件操作支持 …… 76
3.1.1 建立、打开和关闭文件 …… 77
3.1.2 文件的读写操作 …… 78
3.1.3 用于文件操作的可视化组件 …… 81
3.2 实例创建分析 …… 83
3.3 界面风格：TTreeView 和 TListView …… 84
3.3.1 树视图组件 TTreeView …… 84
3.3.2 列表视图组件 …… 87
3.3.3 创建范例程序界面 …… 88
3.4 文件管理和浏览 …… 89
3.4.1 初始化工作 …… 89
3.4.2 树视图的组织和显示 …… 90
3.4.3 列表视图的组织和显示 …… 92
3.4.4 用户浏览命令的响应 …… 96
3.5 实现文件操作功能 …… 98
3.5.1 文件的拷贝、剪切、删除 …… 99
3.5.2 Win32 风格文件重命名的实现 …… 101
3.5.3 文件属性的检视与修改 …… 102
3.6 文件流和内存流 …… 104
3.6.1 文件流（TFileStream 类与 THandleStream 类） …… 104
3.6.2 内存流（TMemoryStream 类） …… 105
3.6.3 其他流式对象 …… 107
3.7 文件相关的高级话题 …… 109
3.7.1 文件加锁和解锁 …… 109
3.7.2 Shell API …… 111
3.7.3 遍历外壳名空间 …… 111
3.7.4 使用 SHBrowseForFolder 函数和 SHFileOperation 函数 …… 117
第 4 章 可与 ACDSee 媲美的 BCBSee32——深入图像文件编程 …… 122
4.1 图像显示技术 …… 124
4.1.1 Windows 图形设备接口 …… 124
4.1.2 TImage 组件 …… 125

4.2 现实图形对象 ……127
4.2.1 TGraphic 类 ……127
4.2.2 TBitmap 类 ……128
4.2.3 TIcon 类和 TMetafile 类 ……129
4.3 使用和控制 JPEG 格式图像 ……132
4.3.1 功能强劲的 TJPEGImage 类 ……132
4.3.2 TJPEGImage 应用示例 ……134
4.4 实例创建分析 ……136
4.5 创建程序界面及浏览窗体部分的实现 ……137
4.5.1 创建程序界面 ……137
4.5.2 浏览系统（Browser）实现 ……138
4.5.3 预览显示处理 ……141
4.6 实现观察窗体部分 ……142
4.6.1 为图像量身定做窗体 ……142
4.6.2 Viewer 窗体中的图像浏览、幻灯功能 ……144
4.6.3 全屏显示和放大、缩小显示 ……145
4.7 图像格式转换和图像打印 ……147
4.7.1 将图像转换为 Bitmap 格式 ……147
4.7.2 将图像转换为 JPEG 格式 ……149
4.7.3 图像打印输出 ……150
4.8 实现特色功能 ……152
4.8.1 设置墙纸 ……152
4.8.2 放大镜的实现 ……153
第 5 章 图像编辑软件 BCB 画板——数字图像处理和图像编辑 ……156
5.1 TCanvas 画布类 ……158
5.1.1 TCanvas 类的重要属性和方法 ……158
5.1.2 TPen、TBrush 和 TColor ……160
5.1.3 重画问题 ……162
5.2 实例创建分析 ……163
5.3 图像编辑程序框架 ……164
5.3.1 创建应用程序界面 ……164
5.3.2 使用光标 ……165
5.3.3 工具箱和颜料盒的实现 ……167
5.4 图像绘制——画图功能的实现 ……168
5.4.1 铅笔、画刷和橡皮 ……169
5.4.2 颜料桶和喷枪 ……171
5.4.3 放大缩小图像、绘制文字 ……172
5.4.4 规则图形的绘制 ……173
5.5 区域选择和图像的剪贴、复制 ……177

5.5.1 区域选择的实现 ……177
5.5.2 应用剪贴板 ……178
5.6 新建、打开、存储文件及简单图像处理 ……181
5.6.1 新建、打开、存储文件 ……181
5.6.2 尺寸设置、反色及图像打印 ……183
5.7 图像处理高级话题 ……185
5.7.1 提升速度 ……185
5.7.2 图像色彩调整 ……188
第 6 章 多样 Windows 屏幕保护程序——动画技术与图形技巧显示 ……194
6.1 Windows 动画技术 ……196
6.1.1 双缓冲区（Double Buffer） ……196
6.1.2 TPaintBox 组件和 TTimer 组件 ……196
6.1.3 生成高性能动画 ……197
6.1.4 掩图技术 ……200
6.2 实例创建分析 ……202
6.3 实现屏幕保护程序框架 ……203
6.3.1 获取并处理应用程序参数 ……203
6.3.2 消息映射 ……204
6.4 动画和特技显示 ……208
6.4.1 屏保的动画部分 ……208
6.4.2 技巧显示 ……212
6.4.3 音乐播放功能 ……216
6.5 屏保设置部分的实现 ……217
6.5.1 存取文件列表 ……218
6.5.2 使用注册表 ……218
6.6 动画技术的其他话题 ……222
6.6.1 桌面精灵动画 ……222
6.6.2 逐帧动画 ……224
6.6.3 多媒体定时器 ……225
6.6.4 高级动画 ……227
第 7 章 完美的多媒体播放器——深入多媒体技术 ……228
7.1 多媒体技术探秘 ……230
7.1.1 多媒体技术的核心 ……230
7.1.2 Windows 操作系统的多媒体服务 ……230
7.1.3 C++ Builder 的多媒体编程 ……232
7.2 多媒体相关组件和多媒体编程 ……234
7.2.1 多媒体 TMediaPlayer 组件 ……234
7.2.2 动画组件 TAnimate ……237
7.2.3 多媒体编程的一般原则 ……238

7.3 实例创建分析 ······239
7.4 媒体播放部分的实现 ······240
7.4.1 基本媒体播放控制 ······240
7.4.2 视频播放相关处理 ······243
7.5 其他关键问题处理 ······245
7.5.1 数字显示实现——使用资源文件 ······245
7.5.2 播放时间进度显示 ······247
7.5.3 实现无标题面板的拖动 ······248
7.5.4 实现音量调整功能 ······249
7.6 MCI 与高级多媒体性能 ······250
7.6.1 TMediaPlayer 组件 ······250
7.6.2 命令消息接口与 mciSendCommand 语言 ······251
7.6.3 播放文件和录制声音 ······253
7.7 底层多媒体 API ······256
7.7.1 RIFF 文件 ······256
7.7.2 使用低级 API 实现 Wave 播放 ······257
第 8 章 俄罗斯方块游戏——VCL 游戏编程与实用技术 ······265
8.1 实例创建分析 ······267
8.2 实现俄罗斯方块程序的核心部分 ······269
8.2.1 程序策划 ······269
8.2.2 数据处理和定制窗体 ······270
8.3 工作模块具体实现 ······273
8.3.1 核心工作模块 ······273
8.3.2 其他问题 ······283
8.4 实用技巧 ······284
8.4.1 创建帮助系统 ······285
8.4.2 使用 INI 文件 ······290
8.4.3 溅出屏幕（Splash Screen） ······292
8.5 为游戏程序增加手柄支持 ······294
第 9 章 快速文件处理工具——进程和多线程技术 ······298
9.1 进程和进程创建 ······300
9.1.1 进程存储 ······300
9.1.2 进程创建方法 ······301
9.1.3 后台进程：制作 Windows 版的 ARJ 工具 ······303
9.2 Win32 多线程技术 ······305
9.2.1 C++ Builder 中实现多线程 ······306
9.2.2 TThread 类 ······307
9.3 实例创建分析 ······308
9.4 实现多线式文件处理工具的技术要点 ······309

9.4.1 主界面线程 ……309
9.4.2 查找线程 ……311
9.4.3 与 VCL 同步 ……313
9.4.4 线程的终止 ……314
9.4.5 拷贝线程 ……315
9.5 多线程调度和线程通信 ……315
9.5.1 优先级和调度 ……315
9.5.2 TEvent 与线程通信 ……317
9.6 多线程高级话题 ……319
9.6.1 对线程计时 ……319
9.6.2 线程本地存储 ……320
9.6.3 线程同步问题 ……321
第 10 章 HTML 浏览器——Internet 相关技术 ……328
10.1 HTTP 协议和 CppWebBrowser 组件 ……330
10.1.1 使用 CppWebBrowser 组件 ……330
10.1.2 使用 NMHTTP 组件 ……332
10.2 创建 BCB WebBrowser 浏览器程序 ……336
10.2.1 CoolBar 工具栏 ……336
10.2.2 实现 Web 页的显示和浏览功能 ……339
10.2.3 辅助功能实现 ……342
10.3 BCB WebBrowser 的第 2 版本……345
10.3.1 安装 ActiveX 控件……346
10.3.2 使用 WebBrowser 控件 ……347
10.3.3 使用文件传输协议（FTP） ……348
10.3.4 邮件协议和其他特定协议 ……351
第 11 章 网页留言簿系统——服务器端 Web 编程 ……354
11.1 生成 HTML 页面 ……356
11.1.1 使用 PageProducer 组件 ……356
11.1.2 在 Web 页发布数据库 ……358
11.2 创建动态 Web 内容 ……362
11.2.1 标准 CGI 编程 ……362
11.2.2 利用 WebModules 技术创建服务器程序……364
11.2.3 实现网站计数器程序……367
11.3 创建基于 ISAPI 的留言簿系统 ……369
11.3.1 ISAPI 编程概述 ……370
11.3.2 在 C++ Builder 中创建 ISAPI DLL ……371
11.4 实现留言簿填写模块 ……372
11.4.1 获取用户输入信息……372
11.4.2 与数据库连接……374

11.5 实现留言簿浏览模块 ……376
11.5.1 显示留言列表……377
11.5.2 显示留言簿详细内容……379
11.6 关于服务器端编程的进一步讨论 ……380
11.6.1 QueryTableProducer 组件 ……380
11.6.2 在线考试/问卷系统 ……381
11.6.3 聊天室系统……382
第 12 章 网络五子棋——WinSock 编程……384
12.1 WinSock 编程概述 ……386
12.1.1 建立服务器端 Socket……387
12.1.2 建立客户端 Socket……387
12.1.3 操纵 Socket 对象传输数据 ……387
12.2 实例创建分析 ……390
12.3 实现网络五子棋程序 ……391
12.3.1 游戏前期工作 ……392
12.3.2 实现联机游戏系统 ……395
12.3.3 简单的辅助功能 ……401
12.4 WinSock 编程高级话题 ……401
12.4.1 流类数据传输 ……401
12.4.2 利用 WinSock 定制协议 ……402
12.4.3 在阻塞状态下传输数据 ……406
12.5 使用 TPowerSock 组件类……406
12.5.1 TPowerSock 组件 ……407
12.5.2 TNMStrm 和 TNMStrmServ 组件……408
第 13 章 图像时钟组件——创建 VCL 组件 ……411
13.1 C++ Builder 组件和组件包 ……413
13.1.1 扩展 VCL 组件 ……413
13.1.2 创建组件的原则 ……413
13.1.3 组件包 ……414
13.1.4 创建一个简单的组件 ……414
13.2 组件编程 ……418
13.2.1 创建组件的起点 ……418
13.2.2 链接图像组件 ……420
13.2.3 编写组件代码 ……422
13.2.4 创建事件 ……424
13.3 创建图像时钟组件 ……426
13.3.1 为组件增加枚举类型属性 ……429
13.3.2 绘制时钟 ……430
13.3.3 增加 TPersistent 属性 ……434

13.3.4 增加新创建的事件 ……435
13.3.5 组件面板位图 ……437
13.3.6 测试 TCoolClock 组件 ……438
13.4 创建非可视化组件 ……439
13.4.1 创建 TOpenDirDialog 组件 ……440
13.4.2 使用非可视化组件 ……442
第 14 章 系统环境监视程序——DLL 及应用 Windows API 编程 ……444
14.1 关于 DLL ……446
14.1.1 在 C++ Builder 中创建 DLL ……447
14.1.2 使用 DLL 实现窗体重用 ……449
14.2 实例创建分析 ……451
14.2.1 理解 Windows API ……451
14.2.2 程序分析 ……452
14.3 编写任务栏指示区图标支持 ……453
14.4 利用 API 实现特殊风格的标题栏 ……456
14.4.1 自绘标题栏 ……456
14.4.2 实现标准标题栏功能 ……459
14.5 窗口与程序 ……460
14.5.1 获得当前所有窗口 ……460
14.5.2 动态汉化窗口菜单 ……462
14.5.3 获得当前激活的进程 ……465
14.5.4 查看/删除系统启动程序 ……467
14.5.5 杀除进程 ……469
14.6 系统与设备 ……470
14.6.1 获取和设置驱动器信息 ……471
14.6.2 获取内存资源信息 ……474
14.6.3 获取设备信息与动态调整显示 ……476
14.6.4 获取和设置系统环境变量 ……479
第 15 章 BCB 抓图大师——高级 DLL 技术和钩子函数 ……481
15.1 DLL 彻底研究 ……483
15.1.1 动态加载 DLL ……483
15.1.2 DLL 入口点及生存周期 ……485
15.2 插件（Plug-In）技术 ……487
15.2.1 插件技术分析 ……487
15.2.2 插件程序实例 ……488
15.3 实例创建分析 ……492
15.4 钩子（Hook）函数 ……493
15.4.1 Windows 的钩子函数 ……494
15.4.2 使用钩子函数的问题 ……495

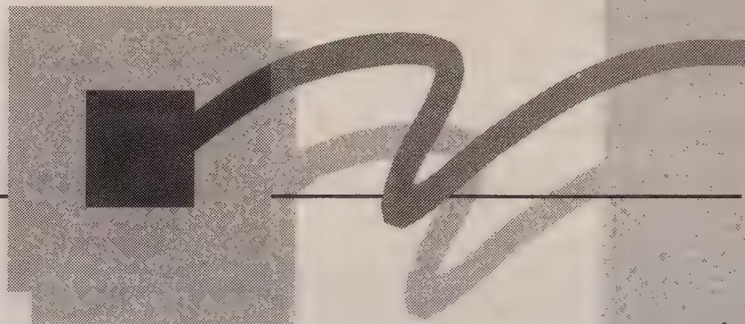
15.4.3 键盘钩子 ……496
15.5 进程间数据共享 ……500
15.5.1 利用内存映像文件共享数据 ……500
15.5.2 在 DLL 中实现存取全局内容代码 ……502
15.6 截图程序的具体实现 ……503
15.6.1 全局存取内存区域的数据组织 ……504
15.6.2 抓图设置处理 ……506
第16章 COM 对象、自动化和 XCoolClock 控件——组件对象模型（COM） ……509
16.1 理解 COM 接口及其实现 ……511
16.1.1 关于 COM 基本概念 ……511
16.1.2 在 DLL 实现类 ……513
16.2 实现 COM 对象 ……516
16.2.1 COM 的服务程序类型 ……517
16.2.2 创建 COM 对象 ……517
16.2.3 创建客户程序 ……522
16.3 几个关键问题 ……524
16.3.1 GUID、CLSID 和 IID ……524
16.3.2 IUnknown 接口 ……525
16.3.3 类工厂（ClassFactory） ……526
16.4 IDispatch、双重接口及 dispinterface ……526
16.4.1 创建 Automation 对象 ……526
16.4.2 创建调用 Automation 对象的客户程序 ……530
16.4.3 IDispatch 和双重接口 ……534
16.5 实现 Word 和 Excel 自动化 ……535
16.5.1 使用 Variant 进行自动化 ……536
16.5.2 自动化 Excel ……540
16.5.3 内部自动化应用程序 ……543
16.6 Internet Explorer 控件的高级用法 ……546
16.6.1 类型库（Type Library） ……546
16.6.2 BCB WebBrowser 的第 3 版本 ……548
16.7 ActiveX 技术和创建 ActiveX 控件 ……550
16.7.1 创建 TCoolClock 的 ActiveX 版本 ……551
16.7.2 为 ActiveX 控件添加属性 ……554
16.7.3 为 ActiveX 控件编写属性页 ……556
16.7.4 ActiveForm 方法 ……558
第 17 章 DirectX 下的 Block 游戏——DirectX 编程 ……562
17.1 DirectX 技术及 DirectX 编程概述 ……564
17.1.1 DirectX 的组成 ……564

17.1.2 DirectX 编程方式 ……565
17.2 使用 DirectDraw ……566
17.2.1 强劲的 DirectDraw 技术 ……566
17.2.2 建立简单的 DirectDraw 程序 ……566
17.2.3 DirectDraw 编程问题……570
17.3 实例创建分析 ……574
17.3.1 程序架构 ……575
17.3.2 处理位图资源 ……575
17.3.3 Block 工程说明 ……576
17.4 具体实现 Block 游戏 ……576
17.4.1 深入 DirectDraw：调色板和位图对象……577
17.4.2 构造 TRing 类：绘制 DirectDraw 位图 ……579
17.4.3 游戏的启动部分 ……582
17.4.4 游戏进行部分 ……583
17.4.5 实现规则 ……588
17.4.6 最后的工作——释放对象 ……590
17.5 DirectX 技术的其他部分……592
17.5.1 使用 Direct3D……592
17.5.2 使用 DirectSound ……595

第 1 章

C++ Builder 5 深入剖析

开发工具概述及特色分析
可视化组件库高级话题
抽象数据类型
使用新版本的 VCL 增强编程



本章导读

Borland C++ Builder 的魅力令无数程序开发人员为之倾倒：它是灵活高效的 C++语言和随心所欲的可视化开发完美结合的产物，它的出现使开发人员不必在高效的底层控制与轻松的程序设计间做出艰难抉择。

本章从开发工具特色及编程理念两方面对 Borland C++ Builder 进行了系统的分析，使你对 C++ Builder 这一优秀开发工具有更加深刻的认识。本章深入讨论了 Visual Component Library 的高级话题和 C++ Builder 常用抽象数据类，同时介绍如何使用 C++ Builder 5 引入的新增技术编程。

本章内容包括：

- Borland C++ Builder 开发工具的概述及分析。
- 可视化组件库（VCL）的高级话题。涉及对 TObject 类、TAppliction 类和 TForm 类的深入分析，以及使用 TMetaClass 实现类引用。
- C++ Builder 中定义的抽象数据类型。包括如何使用链表（TList）、串（AnsiString）、集合（Set）和流（TStream）的内容。
- Borland C++ Builder 5 的 VCL 增强编程。包括使用新增组件 TMonthCalender 和 TActionList、停驻（Docking）技术编程和使用动态数组（DynamicArray）。

1.1　C++ Builder——伟大的开发工具

随着 Windows 95/98、Windows NT 操作系统的出现，我们进入 Win32 编程时代。32 位 Windows 漂亮的图形用户界面（GUI）、多方的多媒体服务、几乎无限制的内存资源、多任务并发执行机制促使用户以及程序开发人员迫不及待地加入 Win32 的行列，编写 Windows 图形界面应用程序。软件工程的革新被 32 位 Windows 操作系统的卓越性能进一步推向深入。

但是，问题也随之而来，Windows 程序更加复杂，规模也更加庞大，从而使开发应用程序的难度加大了。虽然 Microsoft 将各种低级艰深的操作封装到 Windows API 上，但即使这样，编写 Windows 程序仍然需要面对大量与程序主题无关的代码。例如，使用 C++及 Win32 API 函数编写一个无任何功能的完整窗口，需要约 50 行的代码。繁重的代码编制工作使编写美观高效的 Windows 程序的乐趣大大降低了。

在这种情况下，Visual Basic、Delphi 等开发工具因引入了快速应用程序开发（Rapid Application Development，RAD）而大行其道。Visual Basic、Delphi 通过引入控件（Controls）或组件（Components）使得开发 Windows 应用程序变得非常容易，建立复杂 Windows 图形界面如同搭积木一样。特别是 Delphi，它基于 Object Pascal 语言，不仅便捷、灵活、功能强大，而且在执行程序的效率方面也非常出色。Delphi 的出现使得 Windows 编程的元老——Visual C++和 Borland C++黯然失色，也让无数 C++ 程序员羡慕不已。

然而，Borland 公司有这样一个传统：它经常推出具有革命意义的产品。现在，Borland C++ Builder 的推出沿袭了这一传统，横空出世的 Borland C++ Builder 是一个基于真正 C++语言的 RAD 开发工具。Borland 公司天才地将当今最为强劲高效的 C++语言和随心所欲的可视化开发完美结合，产生了新一代的可视化开发工具 Borland C++ Builder。基于组件的编程模式进一步拓展了 C++的能力，使之更强大、更灵活。同时，C++ Builder 继承了 C++语言包括 Visual C++、Borland C++等开发工具的优点，在底层控制方面表现出色。在某种程度上，Borland C++ Builder 可以说是“完美”的，它使得开发人员不必在高效的底层控制与轻松的程序设计间做出艰难的抉择。

Borland C++ Builder 为 C++语言引入了组件编程。组件是今后编程的基础，从现今 Microsoft 大力推广 COM 和 ActiveX 即可看出这一点。C++ Builder 中嵌入了 Delphi 中所使用的高效率的 VCL（Visual Component Library 可视化组件库），VCL 相当出色，Borland C++ Builder 应用 VCL 组件所创建的 Windows 程序至少可以与 OWL 和 MFC 创建的程序一样快、一样小，然而却容易得多。

1. 为什么使用 VCL 组件

Borland C++ Builder 是与 Delphi 一样出色的 RAD 工具。它所使用的是和 Delphi 一样的 VCL 组件库，VCL 是一个完全面向对象的库。它非常强大，并且具有高效率。使用 VCL 组件编程至少可以带来以下好处：

- VCL 组件是封装的产物，封装是一种非常好的编程习惯，OPP 思想的最大好处就是

使封装变得更加简单和有序。VCL 组件出色地封装了 Windows 编程元素，使用组件重用这些封装的代码既正确又快捷。

- VCL 组件的数据隐藏特性提高了编程效率，这样有助于隐藏组件关键的数据结构并且提供正确和抽象的公共接口。
- VCL 组件提供一种明了并且易用的组件模式。它可以像一般的基于 OPP 的编程技术一样快速，并且不容易产生错误。
- VCL 组件模块本身具有极高的效率。实际上，在很多时候，标准的 VCL 对象与同样的 MFC 对象相比更小，速度更快。较之 ActiveX 控件，VCL 的优势则更加突出，虽然在某些方面 ActiveX 可能比 VCL 的功能更强大，但它明显地更庞大、更复杂、速度也更慢。
- VCL 组件使得 RAD 成为可能。使用组件的开发方法和传统的使用 OPP 面向对象开发方法相比，开发周期缩短了 2～4 倍。

VCL 堪称目前设计最为出色的类库。VCL 是真正可视化的，这一点与 MFC 或 OWL 存在本质不同。在 VCL 面前，MFC 或 OWL 这样基于 FrameWork 的类库毫无任何优势可言。例如 Visual C++中的 MFC，虽然 MFC 比 VCL 更为全面，但很多 MFC 类只是将相应的 Win32 API 做了形式上的简单封装，这时候使用 MFC 并不能给开发人员带来太大方便，甚至不如直接调用 API。而在 VCL 所涉及的领域，VCL 的方便程度是 MFC 远远不能比及的。

2. C++ Builder 是 Delphi 的 C++版本吗

经常听到 Delphi 的使用者这样评论 Borland C++ Builder，他们称“C++ Builder 只不过是 Delphi 的 C++版本”。这样的说法在某种意义上并没有错，因为 C++ Builder 实现可视化的核心正是 Delphi 的 VCL 可视化组件库，但问题是，Delphi 的 C++版就一定和 Delphi 能力相同或者比 Delphi 差吗？并不尽然。Borland C++ Builder 是真正的 C++语言的开发工具，C++语言本身的众多优势使 Borland C++ Builder 在很多地方表现得比 Delphi 更为出色。

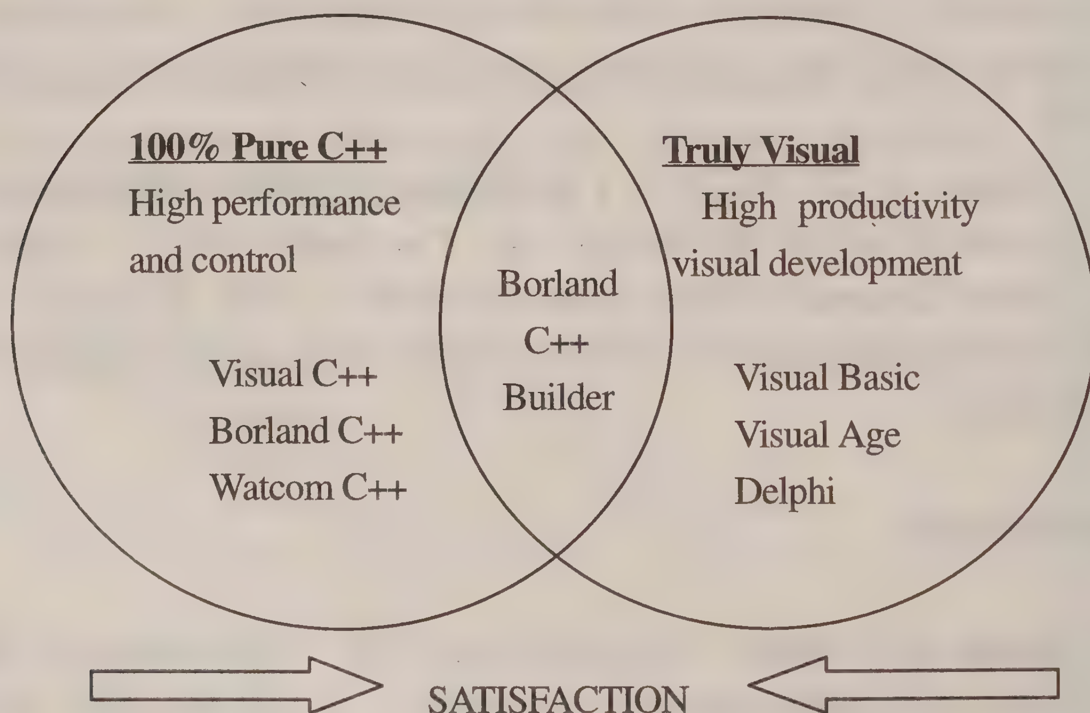


图 1-1　Borland C++ Builder——纯正的 C++语言和可视化技术结合的产物

在这个问题上，我们还是来看看 Borland 公司自己是如何给 Borland C++ Builder 定位的。

图 1-1 是由 Borland 提供的，可以看到，Borland 是要把百分之百纯正的 C++语言和优秀的可视化技术结合起来，这样产生的 Borland C++ Builder 能够在高性能的执行效率与底层控制和快速可视化开发两方面都表现出色，让各种需求的用户满意。

可见，Borland C++ Builder 和 Delphi 的定位并不相同，Borland C++ Builder 的应用领域较 Delphi 更加广泛，能力更强。所以，“Borland C++ Builder 是结合了 VCL 的可视化 C++开发工具”的定位可能更为确切一些。

3. C++语言的魅力

C++语言是当今使用最为广泛的语言，特别是在商业软件及系统软件开发领域，几乎是 C++语言一统天下。这并不是没有原因的。C++语言继承了 C 语言的特点，简洁、灵活而又高效。具体来说，C/C++语言具有以下优点：

- 基本组成元素紧凑、简洁。
- 提供某些接近汇编语言的功能。包括与地址密切相关的强大的指针功能，以及低级的位运算符。能够用高级语言的形式对系统进行某些低级操作，具有强有力的操作硬件接口以及系统设计能力。
- 生成目标代码质量高，程序执行效率高。几乎可以与汇编程序生成的目标代码的效率相同。
- 丰富全面并且正规的数据类型。C++语言提供的数据类型是最为丰富的，C/C++中的数据类型结构，大多已经成为实际上的标准，例如以'\0'结尾的字符串，4B、8B 的浮点类型。实际上，Delphi 正不断扩展 Pascal 的数据类型以和 C/C++保持一致。
- 精致全面的面向对象特性。所有面向对象特性，包括多重继承、多态等 C/C++语言都可支持。

对于 Windows 编程来说，C/C++语言具有天生优势。这是因为 Windows 操作系统的大部分是由 C/C++语言写成的，Windows API 也是为 C/C++程序员提供的编程接口，它仅仅支持 C/C++语言的直接调用。

既然 C++ Builder 使用组件编程，调用 Windows API 还有意义吗？答案是肯定的。虽然 VCL 对 Windows API 做了很出色的封装，但实际上，VCL 完全无法覆盖 Win32 标准 API 和扩展 API 的 2300 多个函数的功能。为了实现一些特殊或者高级的功能，我们不得不使用 API 函数。对于 C++程序员来说，使用 Windows API 正是我们的看家本领。

C++ Builder 以 C++语言为基础，所以相对于 Visual Basic、Delphi 来说，在使用各种强大的 API 方面相当方便。包括庞大的 Win32 API、现下非常流行的 DirectX、OpenGL 等，在 C++ Builder 中都可以毫无困扰地直接调用，同时也完全不用像在 Visual Basic、Delphi 中因为数据类型不一致和声明困难而烦恼，因为就目前的情况来看，所有的 API 接口都是针对 C/C++语言的。

虽然，C++ Builder 提供了丰富的组件，但我们不应该（往往也不能）丢掉 C/C++语言使用 Windows API 编写 Windows 程序的传统。结合方便快捷的 VCL 组件与各种强大 API，将可以胜任任何一个 Windows 编程任务。

4. 出色的新一代开发工具 Borland C++ Builder 5

Borland C++ Builder 5 是一个伟大的开发工具，它是 Inprise 公司（原 Borland 公司）新一代具有战略意义的产品。上文已通过对 C++ Builder 的理解对这个开发工具的功能做了概括的分析。下面将具体介绍 Borland C++ Builder 5 的优点及新特点。

新版本的 Borland C++ Builder 5 主要具有以下优点：

- 保持与传统的 C++语言完全兼容，完全符合 ANSI 标准，结合了改进的 Borland C++ 5.5 编译器和 Delphi 5 编译器，编译速度更快、代码效率更高。在 Borland C++ Builder 5 中，进一步改进了调试工具，调试能力更完善、更灵活。
- 提供可视化的编程界面和实现方案，丰富的 VCL 组件（200 种以上），涵盖 Windows 编程的各个方面。
- 新增对 ADO（ActiveX Data Object）的内部支持，从而使用户能够迅速实现对终端用户用于做商业决策的数据的一致性访问。结合 C++Builder 本身的开放式数据组件结构，如借用于 DBGrid、DBEdit 等数据感知组件，用户无须使用 BDE 就可以很快建立数据库应用程序。
- 遵循开放性标准，程序员可以方便地开发新的 VCL 组件，并将其嵌入 C++ Builder 的 IDE 中。
- 面面俱到。Borland C++ Builder 紧跟编程潮流，提供包括 COM、DCOM、ActiveX、CORBA 的设计接口。在 C++ Builder 5 中，更提供了对 C++ ORB Version 4.0 及 COM+的支持。
- 通过 MTDAS3（Multi-Tier Distributed Application Services Suite）技术，提供多层化（Multi-Tier）数据库途径，替代了传统的客户机/服务器体系。可以容易地使用 Borland C++ Builder 创建分布式应用程序。

Borland C++ Builder 的魅力使笔者在短短时间与它建立了深厚感情，这也是促使笔者写这本书的原因。本书将介绍使用 C++ Builder 进行实际编程的方方面面，并涉及很多高级的技术及技巧。希望您可以从本书中获得想要的东西。

1.2 VCL 高级话题

C++ Builder 提供一系列数量很多，并且相当重要的类。其中一部分是组件类，在 C++ Builder IDE 的组件选项板（Components Palette）上可以看到这些类，另一部分则是更为通用的类，包括大量的抽象数据类型和对象。这些类大多属于 VCL 类库，所以，VCL 不仅仅包括组件部分的内容。本节将研究一些通用的类，并进一步讨论 VCL 类库的结构和 VCL 中一些特殊编程技术。

1.2.1 深入 TObject 类

VCL 的核心是类的层次式结构。TObject 是 VCL 中所有对象的基类，它定义了操纵对象

的基本方法。这个系统中的其他类都是 TObject 类的子类，这样形成一个非常复杂的层次式结构，如图 1-2 所示。

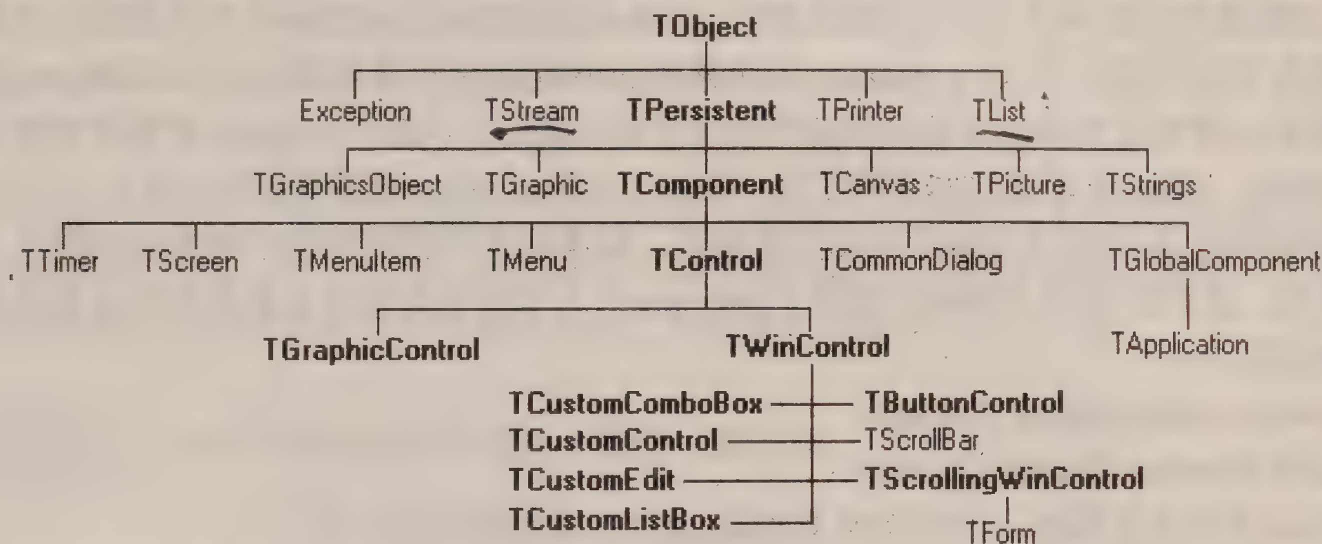


图 1-2 VCL 的层次式结构

图 1-2 表示了 VCL 类库的层次树最初几层中几个非常重要的类。最顶层的是 TObject 类，它是层次式结构中唯一的根类。

TPersistent 是 TObject 的下一级继承者，它是一个抽象类，主要用于提供对象之间的相互赋值和读写流的能力。

TComponent 又是 TPersistent 的继承者，这个类是 VCL 中所有组件的父类，TComponent 定义了组件最基本的行为。对于任何一个 TComponent 类的子类，可以以流的形式存储在 DFM 文件（窗体文件）中（因为 TPersistent 是其父类，该类提供流的实现），并且可以拥有 publish 属性和可视化处理的事件。

TComponent 的一个最为重要的子类是 TControl。继承自该类的组件可以称之为控件，也就是可视化组件。相应地，继承于 TComponent 而不继承于 TControl 的组件是非可视化组件。可视化组件在屏幕上有位置和大小，并且在设计时在窗体上的位置和运行时保持一致。

继承于 TControl 类的子类中的 TWinControl 和 TGraphicControl 非常重要。继承于 TWinControl 的组件是基于系统窗口的可视化组件。它们的共同特点是拥有窗口句柄，并且可以接受输入焦点和包含其他组件。继承于 TGraphicControl 的组件称为图形组件或非窗口组件，这些组件没有窗口句柄，不能接受输入焦点和也不能包含其他组件。非窗口组件只占用较少的系统资源。

此处有一个需要澄清的问题。继承于 TGraphicControl 类的组件不具备输入焦点，但我们清楚地看到像 TLabel、TSpeedButton 这样的非窗口组件具有 OnClick、OnMouseMove 等等事件，似乎和窗口组件无异。实际上，非窗口组件实现 OnClick、OnMouseMove 等事件的原理与窗口组件完全不同。在非窗口组件中，这些与鼠标相关的消息以及其他一些消息，是通过父窗体接收的，并由父窗体发给组件事件。所以，非窗口组件的显示是借助父窗体完成的，而且非窗口组件不可能具有 OnKeyDown 之类的事件。

在本书第 13 章创建 VCL 组件部分中，将进一步讨论与组件类相关的重要基类。现在转回本节的主题——TObject 类。

TObject 是 VCL 类库中所有类的父类，它提供一些对象方法用于返回类信息和一些虚方法能够在其派生类重载。TObject 的特殊地位是它具有某些特殊功能，例如，用户可以使用 TObject 数据类型代替 VCL 中任何数据类型，因为每个类都是从 TObject 类中派生出来的。这个功能的典型应用是，在 C++ Builder 的组件的事件处理函数通常具备一个 TObject 类型的 Sender 参数，通过这个参数可以实现事件处理函数的重用。但是，TObject 类型的变量或参数仅仅能够访问 TObject 定义的方法和属性，例如，如果 Sender 参数的实际引用是一个 TLabel 对象，直接访问 Sender->Caption 将会出现错误。解决这个问题应该使用类的强制转换，在强制转换之前，还可以使用 TObject 类的 ClassNameIs 对象方法来检验参数的实际数据类型。例如下面的代码：

```
if ( Sender->ClassNameIs("TLabel"))
 (TLabel *)Sender->Caption="A Label";
```

TObject 类不具备属性，下面介绍 TObject 类几个有用的对象方法。

- ClassName 方法。

声明：typedef TMetaClass* Tclass;

static ShortString __fastcall ClassName(TClass cls);

ShortString __fastcall ClassName(){ return ClassName(ClassType());}

这个函数返回对象实例的类名。对于 VCL 中任何一个类，可以调用这个基类 TObject 的函数，将返回该类的实际类名。

可以看到，ClassName 对象方法是通过 TClass 类引用实现的。关于类引用的详细内容请参看本章 1.2.4 小节。

- ClassNameIs 方法。

声明：typedef TMetaClass* TClass;

static bool __fastcall ClassNameIs(TClass cls, const AnsiString string);

bool __fastcall ClassNameIs(const AnsiString string){ return ClassNameIs(ClassType(), string); }

这个函数用于判断对象的类型是否是指定的类型。

- ClassParent 方法。

声明：typedef TMetaClass* TClass;

static TClass __fastcall ClassParent(TClass cls);

TClass __fastcall ClassParent() { return ClassParent(ClassType());}

该方法返回对象的父类类型。

- Dispatch 方法。

声明：virtual void __fastcall Dispatch(void *Message);

这是一个虚方法。该方法与实现 Windows 消息机制密切相关。该函数用于调用对象的消息处理方法，实际调用方法由 Message 参数指定的消息来决定。

- InheritsFrom 方法。

声明：typedef TMetaClass* TClass;

static bool __fastcall InheritsFrom(TClass cls, TClass aClass);

bool __fastcall InheritsFrom(TClass aClass) { return InheritsFrom(ClassType(), aClass);}

此函数用于判断两个对象的继承关系。如果 aClass 指定的类是对象的基类，则返回 true，否则返回 false。

- InstanceSize。

声明：typedef TMetaClass* TClass;
static long __fastcall InstanceSize(TClass cls);
long __fastcall InstanceSize(){ return InstanceSize(ClassType()); }
该类返回对象运行时的大小。其效果等同于使用 sizeof 关键字。

下面是一个实例，编写它是用来说明如何使用 TObject 类的对象方法。该实例的关键部分是 ShowObjInfo 函数，其中调用了对象继承于 TObject 类的方法以获得对象信息，包括组件对象的实际类名、实际父类名、对象实例大小以及继承关系等。

Source ShowObjInfo 函数的实现，获得对象信息。

```
//---------------------------------------------------------------------------

void __fastcall TForm1::ShowObjInfo(TObject *Obj)
{
        ListBox1->Items->Add("Class Name: "+Obj->ClassName());
        ListBox1->Items->Add("Parent Class Name: "+
                Obj->ClassParent()->ClassName());
        ListBox1->Items->Add("Object Size:"+
                IntToStr((int)Obj->InstanceSize()));
        if (!Obj->InheritsFrom(__classid(TControl))){
                ListBox1->Items->Add("This is a nonvisual component");
        }
        else
                ListBox1->Items->Add("This is a visual component");
        ListBox1->Items->Add("");
}

//---------------------------------------------------------------------------
```

这段代码演示了 ClassName、ClassParent、InstanceSize 和 InheritsFrom 方法的使用。特别需要说明的是，程序使用中 InheritsFrom 函数判断该组件是可视组件还是非可视组件。这是通过判断该组件是否继承于 TControl 得出的。但在引用 TControl 类时一定要使用__classid 关键字，该关键字是 C++ Builder 的扩展关键字，用于得到一个指定类名在虚表（VTable）的指针。该关键字与类引用相关。

图 1-3 为实例程序的运行结果，图中显示了窗体对象和窗体中组件对象的信息，其中包括非可视组件 TOpenDialog。

可以看到，TObject 类具有很多优异特性。在多数时候，当需要创建一个新类，从 TObject 类继承会比直接创建效果更好，这样做可以使所创建对象自动获得许多有用特性。

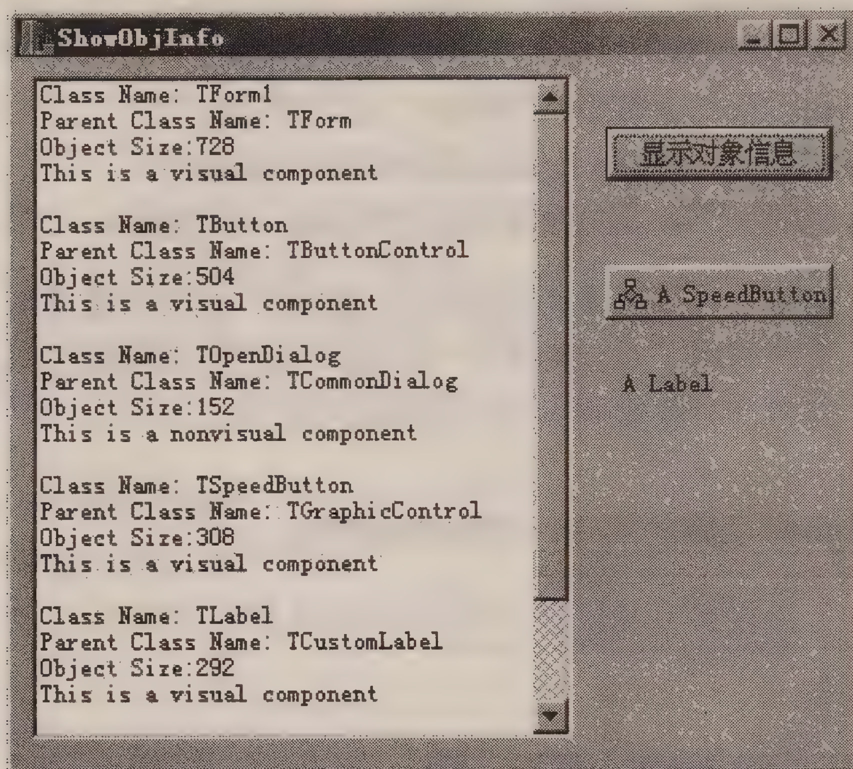


图 1-3　利用 TObject 类的方法获得对象信息

1.2.2　深入 TApplication 类

TApplication 类用于封装 Windows 应用程序对象。TApplication 类包含一系列属性和方法反映 Windows 应用程序的一些基本操作，包括应用程序的创建、运行、维持和销毁。TApplication 类提供了一个在开发者和 Windows 环境间的简单界面。具体来说，TApplication 类提供以下的功能：

- 与应用程序的 Windows 消息处理。
- 上下文相关的在线帮助支持。
- 菜单的加速表和热键的实现。
- 异常处理。
- 对于 Windows 程序基本实现部分的管理。

TApplication 类并不出现在 C++ Builder 的组件选项板上，尽管它继承于 TComponent 类，并且一般情况不能自己声明一个 TApplication 类的对象。每一个应用程序都会自动创建一个 Application 全局变量，该变量是 TApplication 类的一个实例。通过 Application 变量，编程人员可以进行与应用程序相关的一系列重要的操作。

TApplication 类具有以下一些重要的属性、方法和事件。

1. TApplication 类的重要属性

- Active 属性。

声明：__property bool Active = {read=FActive, nodefault};

当应用程序激活时，该属性为 true，否则为 false。一旦应用程序窗体失去焦点，Active 属性返回 false。

- ExeName 属性。

声明：__property System::AnsiString ExeName = {read=GetExeName};

该属性可以获得应用程序可执行文件的文件名。

- HintColor 属性。

声明：__property Graphics::TColor HintColor = {read=FHintColor, write=SetHintColor, nodefault};

这个属性可以指定应用程序中暗示文本（Hint）的背景颜色。

- HelpFile 属性。

声明：__property System::AnsiString HelpFile = {read=FHelpFile, write=FHelpFile};

指定与应用程序相连的帮助文件。

- Icon 属性。

声明：__property Graphics::TIcon* Icon = {read=FIcon, write=SetIcon};

提供一个标识应用程序的对话框，在运行期间对这个属性赋值会造成任务栏上应用程序图标的改变。在设计期间必须通过【**Project/Options**】菜单指定应用程序图标。

- ShowMainForm 属性。

声明：__property bool ShowMainForm = {read=FShowMainForm, write= FShowMainForm, nodefault};

决定当应用程序执行时是否显示主窗体。有时候可能会需要在程序开始运行时隐藏窗体，而等待以其他的方式将窗体激活。

- Title 属性。

声明：__property System::AnsiString Title = {read=GetTitle, write=SetTitle};

应用程序标题。标题会在任务栏上显示。在设计期间必须通过 Project|Options 菜单指定应用程序标题。

2. TApplication 类的重要方法

- HandleMessage 方法。

声明：void __fastcall HandleMessage(void);

调用 HandleMessage 方法可以暂时中断应用程序执行并使 Windows 处理消息队列中的一条消息。如果消息队列为 NULL，HandleMessage 将产生一个 OnIdle 事件。

- Minimize 方法。

声明：void __fastcall Minimize(void);

使应用程序收缩到 Windows 任务栏。

- ProcessMessages 方法。

声明：void __fastcall ProcessMessages(void);

调用 ProcessMessages 方法中断应用程序执行并处理当前消息队列的消息，直到消息队列为 NULL 为止。ProcessMessages 不允许应用程序出现空闲（Idle），这点和 HandleMessage 不同。

3. TApplication 类的重要事件

Application 对象的事件具有一定的特殊性，因为它不能像组件那样在设计时指定处理事件

的函数，而只能在程序中进行设定。

- OnHint 事件。

当鼠标指针移动到某个组件上并可以显示一个暗示文本时发生（此时 Hint 框还未出现）。通过该事件可以定制窗体中各个控件对象 Hint 的显示方式。

- OnIdle 事件。

当应用程序空闲时发生。应用程序已没有可以处理的消息即在等待消息时，会发生此事件。

- OnMessage 事件。

应用程序接收一个 Windows 消息时，该事件发生。

- OnMinimize 事件。

当应用程序最小化时，该事件发生。利用该事件可以轻易实现类似 Winamp 等程序中最小化后在 Windows 任务栏消失，而只保留指示区图标的功能。

下面的例子演示了最为重要的两个 TApplication 类的事件——OnMessage 事件和 OnHint 事件的使用。该程序通过响应 OnMessage 事件的方式处理了 Windows 消息 WM_RBUTTONDOWN，而不是利用 TForm 类的 OnMouseDown 事件。同时，通过处理 TApplication 类的 OnHint 事件实现将所有组件的 Hint 文本都转为在状态条显示。

Source 使用 OnMessage 事件和 OnHint 事件

```
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
   : TForm(Owner)
{
      Application->OnMessage = AppMessage;
      Application->OnHint = DisplayHint;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::AppMessage(tagMSG &Msg, bool &Handled)
{
      if (Msg.message == WM_RBUTTONDOWN )
      {
            MessageBox(Handle,("RButtonDown Message/at Point("
                  +IntToStr(LOWORD(Msg.lParam))+","+IntToStr(HIWORD(Msg.lParam))+").").c_str(),
                  "Message",MB_OK);
            Handled = true;
      }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::DisplayHint(TObject *Sender)
```

```
{
    StatusBar1->SimpleText = GetLongHint(Application->Hint);
}
```

该实例程序的执行结果如图 1-4 所示。

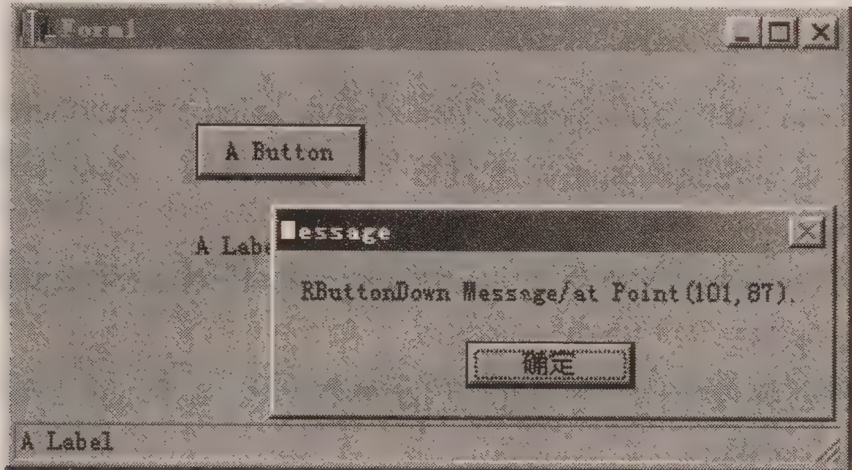


图 1-4　使用 Application 对象的事件，定制 Hint 输出并提供另一种处理消息的方法

1.2.3　深入 TForm 类

TForm 类用于描述 Windows 应用程序中的所有窗体对象，它同样是一个非常重要并且具有特殊性的类。TForm 类继承于 TControl，所以它是可视的，但 TForm 并没有像其他可视组件那样出现在 C++ Builder 的组件选项板上。如果要向程序中添加一个窗体，只能通过菜单或工具栏来完成——选择【**File/New Form**】菜单项或单击工具栏的相应按钮。

TForm 类对应的窗体是 C++ Builder 中各种组件的容器，并且是可视化设计时所有组件的最终容器（一些可视组件，如 Panel、ToolBar 也是组件的容器）。同时，每一个应用程序的窗体（TForm 类的派生类对象）对应一个单元（Unit），C++ Builder 会为每个单元生成对应的头文件和 CPP 文件。

一般来说，没有窗体直接是 TForm 类的实例。在 C++ Builder 中所创建的所有窗体，实际上都是创建了一个继承于 TForm 类的新类，这个新类的定义可以在相应的头文件中看到。实际编程中，程序员所面对的就是一些 TForm 类的派生类，并且会为这个 TForm 类的派生类添加变量成员、属性和方法，以实现窗体所需要的特定功能。

在此必须说明，窗体与窗口是一个不同的概念。窗口是一个广泛的概念。在 Windows 中，用户界面的大多数元素都是窗口。用 VCL 的概念来解释，窗口是继承于 TWinControl 类的各种元素。更具体地说，窗口具有以下的一般特点：

- 具有窗口句柄，Windows 系统可以识别它们，并为窗口引用函数。
- 当系统中有事件发生，消息会发往相应的窗口。每个窗口都具有一个隐含的窗口函数（WindowProc）处理系统发给窗口的各种消息。

窗体也是窗口，但它特指那些一般来说具有标题栏、边框、可以进行最大最小化的窗口。这类窗口也就是用户见到的一般意义的应用程序窗口。所以，窗体是应用程序中必不可

少的重要组成部分。

深入了解 TForm 类需要深入了解该类的重要属性、方法和事件。下面将对 TForm 类一些重要并且容易忽视的属性、方法和事件作以介绍。

1. TForm 类的重要属性

- AutoScroll 属性。

声明：__property bool AutoScroll={read=FAutoScroll, write=SetAutoScroll, default=1};

决定当窗体尺寸不足以完整的显示出所有窗体上的组件时，滚动条是否自动出现。

- BorderStyle 属性。

声明：__property TFormBorderStyle BorderStyle={read=FBorderStyle,write = SetBorderStyle , stored =IsForm, default=2};

指定显示窗体的边界类型。可取值包括 bsNone、bsSingle、bsSizeable、bsDialog、bsToolWindow 及 bsSizeToolWin。

- Canvas 属性。

声明：__property Graphics::TCanvas* Canvas = {read=GetCanvas};

窗体画布。提供进入窗体绘图区域的属性。

- ClientRect 属性。

声明：__property Windows::TRect ClientRect = {read=GetClientRect};

返回窗体的客户矩形区域。注意，窗体客户区域和窗体区域是不同的。客户区不包括窗体边界和标题栏。与客户区相关的属性还包括 ClientHeight 和 ClientWidth。

- ComponentCount 属性。

声明：__property int ComponentCount = {read=GetComponentCount, nodefault};

返回当前窗体上所有组件的个数。

- Components 属性。

声明：__property TComponent* Components[int Index] = {read=GetComponent};

提供通过索引序号访问窗体上的组件。通过 ComponentCount 和 Components 可以实现窗体的所有组件的遍历。TForm 类还具有相应的两个属性 ControlCount 和 Controls，通过这两个属性可以对窗体的所有控件进行遍历。

- FormStyle 属性。

声明：enum TFormStyle = { fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop };

__property TFormStyle FormStyle = {read=FFormStyle, write=SetFormStyle, stored=IsForm, default=0};

决定窗体类型。可以设置窗体为通常窗体（fsNormal）、多文档应用程序主窗体（fsMDIForm）、多文档程序子窗体（fsMDIChild）和总在最前的窗体（fsStayOnTop）。

- MDIChildCount 属性。

声明：__property int MDIChildCount = {read=GetMDIChildCount, nodefault};

返回目前打开的多文档程序子窗体的个数。

- MDIChildren 属性。

声明：__property TForm* MDIChildren[int I] = {read=GetMDIChildren};

通过索引序号访问每一个 MDI 子窗体，提供指向 MDI 子窗体对象的指针。

- Menu 属性。

声明：__property Menus::TMainMenu* Menu = {read=FMenu, write=SetMenu, stored = IsForm};

指定窗体的主菜单，类似还有 PopupMenu 属性用于指定窗体的弹出菜单。

- WindowProc 属性。

声明：typedef void __fastcall (__closure *TWndMethod)(Messages::TMessage &Message);

__property TWndMethod WindowProc = {read=FWindowProc, write=FWindowProc};

指向窗体的窗口函数。使用这个属性可以指定窗体使用自己定制窗体的窗口函数，以实现所需要的特殊消息处理。

- ModalResult 属性。

声明：__property TModalResult ModalResult = {read=FModalResult, write= FModalResult, nodefault};

设定模态窗口的返回值。即使用 ShowModal 方法显示窗体的返回值。

2. TForm 类的重要方法

TForm 类提供几个基本窗体操作方法，包括窗体的关闭（Close）、隐藏（Hide）和显示（Show）等，以及用于 MDI 窗体的子窗体排列操作方法，包括 ArrangeIcons、Cascade 和 Tile 等。除此之外，TForm 类还有以下一些重要方法需要注意。

- ClientToScreen/ScreenToClient 方法。

声明：Windows::TPoint __fastcall ClientToScreen(const Windows::TPoint &Point);

转换一个给定点的坐标。使用 ClientToScreen 将客户区坐标转换为全局屏幕坐标，使用 ScreenToClient 完成相反的工作。

- ControlAtPos 方法。

声明：TControl* __fastcall ControlAtPos(const Windows::TPoint &Pos, bool AllowDisabled);

返回窗体中在指定坐标位置存在的控件。

- FocusControl/DefocusControl 方法。

声明：void __fastcall FocusControl(Controls::TWinControl* Control);

使窗体中的某个控件获得焦点/失去焦点。

- Invalidate 方法。

声明：virtual void __fastcall Invalidate(void);

使得窗体区域失效而重画。

- ScaleBy 方法。

声明：void __fastcall ScaleBy(int M, int D);

设置窗体比例以改变窗体大小。参数 M 和 D 分别为放缩倍数的分子和分母。例如，如果在程序中使用：

```
Form1->ScaleBy(3,4);
```

将使 Form1 窗体的尺寸被 3 乘后再被 4 除，即最后得到的窗体大小是原来的 3/4。

- ShowModal 方法。

声明：int __fastcall ShowModal(void);

以模态窗体的方式显示窗体。该方法可以进行显示并得到返回值。

1.2.4 TMetaClass 和类引用

现在，我们来讨论 VCL 中一个基础的问题——类引用（Class Reference）。理解类引用对深入理解 VCL 的实现机制和 VCL 的公共基类 TObject 类的一些问题很有好处。

有时候一些操作会发生在类的类型本身，而不是类的一个实例（也就是对象）。这时候就可以使用类引用。需要明确的是，类的引用不是实际的类，它不是对象，不是对象的引用，而是对类的类型的引用。

在 C++ Builder 中类的引用通过一个特殊的类型 TMetaClass 实现，这点可以通过 TObject 类的一些对象方法的声明中看到。TMetaClass 用于定义类的引用变量的类型。例如，假设程序中定义了一个类 AClass，则可以编写如下的代码定义一个引用类型并引用该类：

```
TMetaClass *ClassRef;
ClassRef=__classid(AClass);
```

这段代码用到了一个 C++ Builder 的扩展关键字——“_classid”。这个关键字与实现类引用密切相关，简单来说，它可以返回一个类的引用类型。

类引用在某些时候具有非常重要的意义。程序员可以通过类引用来创建对象，另外，在 C++ Builder 中 VCL 的内部实现机制用到了类引用。

举一个常见的例子来说明类引用：每个程序都会使用的 Application 对象的 CreateForm 方法，这个方法用到类的引用作为一个参数。打开任何一个工程的主入口文件（例如 Project1.cpp，而非单元文件的那个 CPP），即可看到类似下面的语句：

```
Application->CreateForm(__classid(TForm1), &Form1);
```

这里，第一个参数是类引用，第二个参数是实际对象。

C++ Builder 在运行时库和 VCL 中声明了许多类的引用，例如：

```
typedef TMetaClass* TClass;
typedef TMetaClass* TCompenontClass;
typedef TMetaClass* TControlClass;
typedef TMetaClass* TExceptClass;
typedef TMetaClass* TWinControlClass;
```

现在你应该可以真正理解在 TObject 类的一些对象方法定义中常常出现的 TClass 类型参数的确切含义。类引用技术为在运行时灵活的处理基于类的数据类型提供了途径。

1.3 抽象数据类型

Borland C++ Builder 以类的形式提供了几个非常有用的抽象数据类型。包括链表类（TList）、字符串类（AnsiString）、集合类（Set）、动态数组类（DynamicArray）和流类（TStream），这些抽象数据类型在相当多的地方起着重要作用。C++ Builder 组件的属性、方法及 C++ Builder 函数的参数常常定义为这些数据类型。

更为重要的是，在 C++ Builder 编程中，程序员也可以使用这些类型定义变量，这些类的强大功能会给编程工作带来很多方便。

1.3.1 链表（TList）

TList 类实现了可以动态存储的线性链表对象。链表是一种非常有用的数据类型，它远比数组灵活的多，因为链表可以在运行时动态的扩展和删除。同时，C++ Builder 的 TList 类还具备以下几个显著的优点：

- TList 类实现通用的线性表对象，它支持任意类对象，包括用户自定义的类。
- 虽然 TList 类实际上以存储指针（链式存储）的方式实现，但它同时提供了顺序存储的查找方式。可以通过 Items 属性像访问一个数组那样访问 TList 对象的每一项。
- 提供方便全面的链表功能。例如，全套插入、删除、查找、排序的对象方法以及包含 Count 属性访问链表中项目的个数。

在 C++ Builder 众多组件的属性中，以 TStrings 类型定义的非常多见。实际上，TStrings 类是继承于 TList 类的一个抽象类，它代表所有字符串形式的链表（字符串列表），但 TStrings 不考虑字符串列表的存取实现。TStrings 类定义的是抽象列表，所以 TStrings 类型仅仅可以定义为可存储字符串列表的组件的属性，如 TListBox 列表框、TMemo 多行文本框等等。通常情况下，不能使用 TStrings 类定义自己的字符串列表，而应该使用 TStringList 类。

TStringList 类是 TStrings 类的子类，它用于程序员自己定义一个字符串列表。该类实现了 SaveToFile 和 LoadFromFile 等等方法，SaveToFile 方法可以自动存储字符串列表的内容到一个指定的文本文件中，而 LoadFromFile 方法提供从文本文件中获取内容到字符串列表中。

下面介绍 TList 类的重要属性和方法。

1. TList 类的重要属性

- Capacity 属性。

声明：__property int Capacity = {read=FCapacity, write=SetCapacity, nodefault};

指定 TList 对象的存储空间，即一次性的为 TList 对象分配内存。当用户已经设定 Capacity 属性，如果没有足够的内存支持，EOutOfMemory 异常将会出现。设置 Capacity 属性可以避免使用 Add 方法或 Delete 方法时频繁的内存分配、释放操作，这样可以提高应用程序的运行效率。例如下面一段代码：

```
List->Clear();
List->Capacity = Count;
for (int I = 0; I < Count; I++)
    List->Add(...);
```

- Count 属性。

声明：__property int Count = {read=FCount, write=SetCount, nodefault};

常常会用到的属性，用于设置和返回列表对象中元素的个数。

- Items 属性。

声明：__property void * Items[int Index] = {read=Get, write=Put};

通过这个属性可以像访问数组那样通过序号访问列表中的元素。

- List 属性。

声明：typedef void *TPointerList[134217727];

typedef TPointerList *PPointerList;

__property PPointerList List = {read=FList};

获得直接进入列表中元素数组的指针。

2. TList 类的重要方法

- Add 方法。

声明：int __fastcall Add(void * Item);

加一个新元素到列表的末尾。

- Clear 方法。

声明：DYNAMIC void __fastcall Clear(void);

清空一个列表对象的所有元素。

- Delete 方法。

声明：void __fastcall Delete(int Index);

删除列表中的一个指定序号的元素。

- Exchange 方法。

声明：void __fastcall Exchange(int Index1, int Index2);

交换列表中的两个元素的位置。

- IndexOf 方法。

声明：int __fastcall IndexOf(void * Item);

查询对象方法。查找列表中具有指定值的元素的位置。

- Insert 方法。

声明：void __fastcall Insert(int Index, void * Item);

插入一个元素到列表的指定位置。

- Sort 方法。

声明：typedef int __fastcall (*TListSortCompare)(void * Item1, void * Item2);

void __fastcall Sort(TListSortCompare Compare);

列表元素的排序方法。使用这个方法首先应定义一个比较函数。

下面通过一个例子来讨论通用 TList 类的使用方法。当用户需要一个任意类型数据的链表时，可以声明一个 TList 对象，并为其增添数据。当要访问数据时，可以对列表元素强制转换，转换为可操作的现实对象。在这个例子中定义了一个 TBug 类，每一个 TBug 类的实例代表一个在窗体上显示的虫子。TBug 类实现了 Show 方法和 Move 方法，用于虫子图像的显示和移动操作。该类的完整定义及实现如下：

Source 在 ListDemo 范例中的 TBug 类定义

```
class TBug : public TObject
{
```

```
private:
        Graphics::TBitmap *FPic;
        int FX;
        int FY;
        TCanvas * FCanvas;
public:
        __fastcall TBug(int AX,int AY,AnsiString FileName,TCanvas *ACanvas);
        void __fastcall Move(int Dir);
        void __fastcall Show();
        __property int Y  = { read = FY, write = FY };
        __property int X  = { read = FX, write = FX };
};
//---------------------------------------------------------------------------
void __fastcall TBug::Show()
{
        FCanvas->Draw(FX,FY,FPic);
}
//---------------------------------------------------------------------------
__fastcall TBug::TBug(int AX,int AY,AnsiString FileName,TCanvas *ACanvas)
{
        FX=AX;
        FY=AY;
        FPic=new Graphics::TBitmap();
        FPic->LoadFromFile(FileName);
        FCanvas=ACanvas;
        Show();
}
//---------------------------------------------------------------------------
void __fastcall TBug::Move(int Dir)
{
switch(Dir){
case 1: // up
        FY-=3;
        break;
case 2: // down
        FY+=3;
        break;
case 3: // left
                FX-=3;
                break;
```

```
        case 4: // right
            FX+=3;
            break;
        }
}
//---------------------------------------------------------------------------
```

ListDemo 范例程序会向一个 TList 对象添加 TBug 类的数据。这样，就可以通过 TList 来方便地管理这组 TBug 类对象，包括添加(Add)、删除(Delete)、清空(Clear)以及操作指定的元素对象（移动 Bug）。程序运行结果如图 1-5 所示。

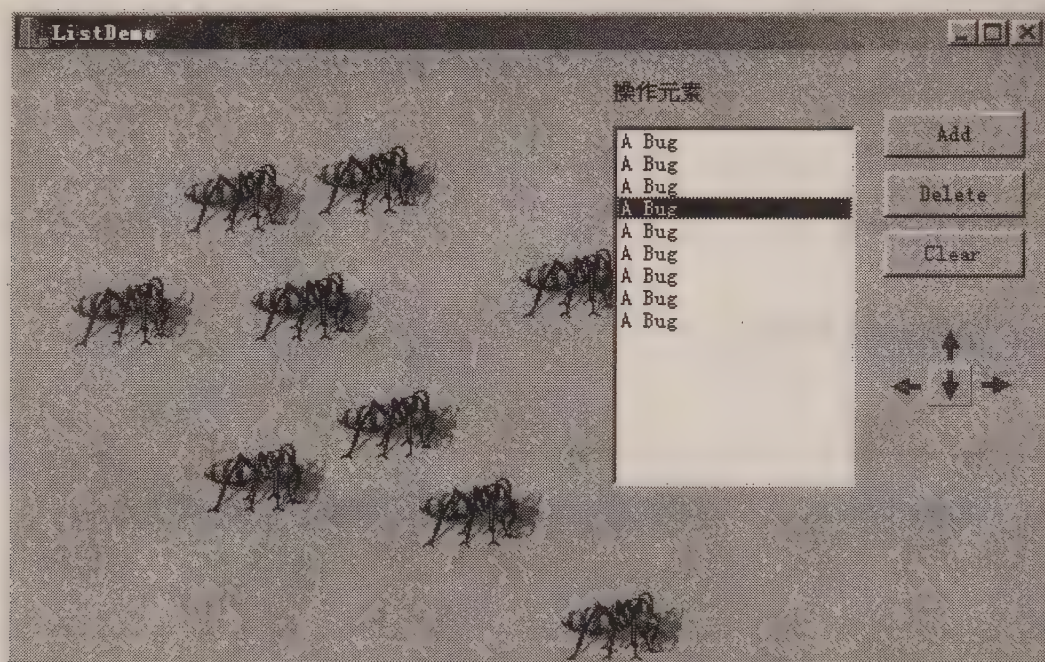


图 1-5　ListDemo 实例程序

Source　使用 TList 类管理自定义对象

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{       // 添加元素
        BugList->Add(new TBug(random(300),random(300),"live.bmp",Canvas));
        ListBox1->Items->Add("A Bug");
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{       // 删除元素
        if (ListBox1->ItemIndex>=0){
        TBug *tmpBug=(TBug *)BugList->Items[ListBox1->ItemIndex];
        delete tmpBug;
        BugList->Delete(ListBox1->ItemIndex);
        ListBox1->Items->Delete(ListBox1->ItemIndex);
```

```
            Invalidate();
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{      //清空列表
        TBug *tmpBug;
        for (int i=0;i<BugList->Count;i++)
        {
            tmpBug = (TBug*)BugList->Items[i];
            delete tmpBug;
        }
        BugList->Clear();
        ListBox1->Items->Clear();
        Invalidate();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormPaint(TObject *Sender)
{      //重画窗体。在这段程序中通过 BugList 对象轻易的遍历了所有元素
        TBug *tmpBug;
        for (int i=0;i<BugList->Count;i++)
        {
           tmpBug=(TBug *)BugList->Items[i];
           tmpBug->Show();
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::SpeedButton1Click(TObject *Sender)
{      //控制某一个 Bug 对象上移的按钮函数
       //类似可以实现下移、左移和右移
       if (ListBox1->ItemIndex>=0){
           TBug *tmpBug=(TBug*)BugList->Items[ListBox1->ItemIndex];
           tmpBug->Move(1);
           Invalidate();
       }
}
//---------------------------------------------------------------------------
```

可以看到，TList 类是非常方便、非常强大的。尽管 C++ Builder 5 引入了动态数组（DynamicArray），但 TList 类较之动态数组仍然要显得灵活和方便的多（TList 类具备的实用对象方法是动态数组所不及的）。动态数组唯一超越 TList 类之处是：动态数组可以定义数

据类型，TList 列表仅仅简单的存储指针，而使用指针可能会造成某些危险。

1.3.2 字符串（AnsiString）

AnsiString 是 C++ Builder 中最常用的字符串类型。VCL 组件中任何用到字符串变量的地方，无一例外地使用了 AnsiString 类型字符串。AnsiString 作为一个类定义，它较之 C/C++语言传统的 char 类型数组实现字符串的方法具有多方面优点，因为 AnsiString 提供了丰富的对象方法，涵盖了字符串处理的方方面面。使用 AnsiString 类型访问、操作和使用字符串都很方便。

本质上，AnsiString 仍然采用 C/C++的字符串结构，即以'\0'字符（NULL）结尾的字符串，所不同的是 AnsiString 类型在实际字符串数据前增加数个字节的信息头。AnsiString 类定义了一系列操作符（Operator），使得使用 AnsiString 较传统的 C/C++字符串容易的多。这些操作符包括：

- 赋值 “=”。
- 连接 “+”、“+=”。
- 比较 “==”、“>”、“<”、“<=”、“>=”、“!=”。
- 访问 “[]”。

AnsiString 另一个突出优点是提供大量实用方法，使复杂的字符串操作可以轻易解决。下面介绍 AnsiString 类的重要方法。

- c_str 方法。

声明：char* __fastcall c_str() const;

返回一个标准 C/C++格式的字符串。这个方法相当常用，因为使用 Windows API 以及其他 API 或库的时候，只能支持标准 C/C++的以'\0'字符（NULL）结尾的字符串。使用 c_str 方式可以进行 AnsiString 到标准字符串的转换。

- Delete 方法。

声明：void __fastcall Delete(int index, int count);

删除字符串中指定数量的字符。

- FormatFloat 方法。

声明：static AnsiString __fastcall FormatFloat(const AnsiString& format,const long double& value);

该方法可以使用掩码格式化一个数字并输出到字符串中。一个典型的应用是给一个很长的数字增加每三位的逗号分隔符。

- Insert 方法。

声明：void __fastcall Insert(const AnsiString& str, int index);

插入一个字符串到原字符串的指定位置。

- IntToHex 方法。

声明：static AnsiString __fastcall IntToHex(int value, int digits);

将一个整数转换为一个 16 进制数字到字符串中。

- Length 方法。

声明：int __fastcall Length() const;

返回字符串的实际长度。

- LoadStr 方法。

声明：static AnsiString __fastcall LoadStr(int ident);

从可执行文件的字符资源中加载字符串。

- LowerCase/UpperCase 方法。

对字符串中的英文字符进行大小写转换。

- Pos 方法。

声明：int __fastcall Pos(const AnsiString& subStr) const;

在原字符串中查找一个子串的位置。

- SubString 方法。

声明：AnsiString __fastcall SubString(int index, int count) const;

返回原字符串的一个指定子串。

- Trim 方法。

声明：AnsiString __fastcall Trim() const;

除去了原字符串的空格、制表符和控制符，并返回一个字符串。

- WideChar 方法。

声明：wchar_t* __fastcall WideChar(wchar_t* dest, int destSize) const;

转换 AnsiString 到一个宽字符数组。因为在 COM 中的字符串是宽字符串，所以该方法在使用 COM 工作时会经常被用到。

1.3.3 集合（Set）

严格来说，C++ Builder 的集合（Set）并不是一个类，而是一个类模板（Class Template）。它用来实现集合这个抽象数据类型。

使用集合类模板可以定义一个实际的集合类型，它的一般声明模式是这样的：

```
typedef Set<type, minval, maxval> ClassName;
```

其中，type 用来指定集合元素的类型；minval 指定集合元素的最小值；maxval 指定集合元素的最大值。例如可以这样定义一个集合：

```
typedef Set<char,'A','Z'> TUpperSet;
```

这行代码定义了一个名为 TUpperSet 的集合类，这个集合类的可能元素为 26 个大写字母（即全集为 26 个大写字母组成的集合）。

然后可以使用这个集合类声明一个具体的集合对象，例如：

```
TUpperSet UpperSet;
```

类模板（Class Template）是 C++语言引入的概念，在 Visual C++和 Borland C++中都有具体应用。如果对类模板不是很清楚，请参看一些讲述 C++语言的书籍。

集合类具有一系列操作符和方法，以下是一些常用操作符和方法的说明。

- Clear 方法。

移除集合对象的所有元素，执行这个方法后集合为空集。

- Contains 方法。

声明：bool __fastcall Contains(const T el) const;

查询集合中是否包含指定的元素。

- operator –。

声明：Set __fastcall operator – (const Set& rhs) const;

两个集合进行求差运算，结果集合包含两个原集合中不相同的元素。

- operator *。

声明：Set __fastcall operator *(const Set& rhs) const;

求两个集合的交集。

- operator +。

声明：Set __fastcall operator +(const Set& rhs) const;

求两个集合的并集。

- operator <<。

声明：Set& __fastcall operator <<(const T el);

添加一个元素到指定集合。

- operator >>。

声明：Set& __fastcall operator >>(const T el);

从集合中移除一个指定元素。

这些操作符和方法在 VCL 中集合类型中也是常常出现的。最典型的一个应用是鼠标事件和键盘事件处理函数的 Shift 参数，该参数是 TShiftState 类型，这个类型就是一个集合类。TShiftState 类是这样声明的：

```
typedef Set<Classes__1, ssShift, ssDouble> TShiftState;
```

另外一个常见应用是在 MessageDlg 函数中，该函数具有一个 TMsgDlgButtons 类的参数 Buttons，TMsgDlgButtons 也是一个集合类。

下面是一段简单的代码，演示了 TShiftState 类、TMsgDlgButtons 类——两个集合类的使用。在这个例子中，当用户同时按下 Alt、Ctrl、S 键时，将会弹出一个 MessageBox，这个 MessageBox 是调用 C++ Builder 的 MessageDlg 生成的，程序执行的结果如图 1-6 所示。

Source 使用集合类型属性

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
    TShiftState Shift)
{
    if (Key=='S'&&Shift.Contains(ssAlt)&&Shift.Contains(ssCtrl))
    {
        TMsgDlgButtons Buttons;
```

```
        Buttons<<mbYes<<mbNo<<mbYesToAll<<mbAbort<<mbHelp;
        MessageDlg("集合类演示程序!",mtConfirmation,Buttons,0);
    }
}
//---------------------------------------------------------------------------
```

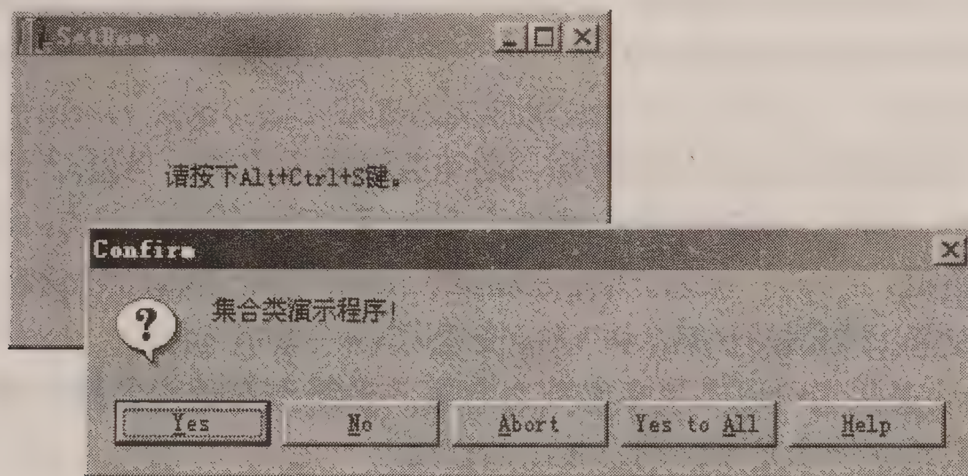


图 1-6　SetDemo 程序演示了 VCL 中集合类型属性的使用

1.3.4　动态数组（DynamicArray）

数组一直是所有高级语言中必不可少的构造类型。然而，在一般的编译型语言中，数组只能在初始化时定义一个确定的大小，并且不能在运行时动态改变数组元素个数。而 C++ Builder 实现了动态数组（DynamicArray），它可以动态地改变数组长度，并且像一般数组那样易于使用。DynamicArray 可以说是 VCL 中的一个重大的革新。

动态数组是以类的方式实现的。C++ Buillder 提供了 DynamicArray 类模板，使用这个类模板可以声明实际动态数组类。使用下面的语法：

```
DynamicArray<type>ArrayName;
```

其中 type 为动态数组的数据类型，动态数组支持任何类型的数据、对象，甚至是一个动态数组对象。例如可以这样声明一个动态数组：

```
DynamicArray<DynamicArray<AnsiString>>aArray ;
```

这相当于声明了一个二维动态数组。

动态数组具有一个 Length 属性，通过这个属性可以设置或获得数组的长度。例如：

```
DynamicArray<int>IntArray;
IntArray.Length=10;
ShowMessage("ArrayLength:"+IntToStr(IntArray.Length))";
```

如果要释放一个动态数组，应该将该动态数组的 Length 属性设为 0。

动态数组的赋值和访问与一般数组相同，因为动态数组实现了“[]”操作符，用于访问数组中的元素。下面的代码演示了动态数组的赋值和访问：

```
for(int i=0;i<=IntArray.Length;i++)
    IntArray[i]=2*i;
```

```
int total=0;
for(int i=0;i<=IntArray.Length;i++)
    total +=IntArray[i]
return total;
```

另外，动态数组具有 Low 和 High 两个属性，分别表示动态数组的起始下标和中止下标。起始下标总是为 0，而中止下标总是等于 Length-1。所以上面的程序也可以这样写：

```
for(int i=IntArray.Low;i<=IntArray.High;i++)
    total+=IntArray[i];
```

动态数组较一般数组更为容易复制数据。动态数组实现了“=”操作符，可以复制整个数组到另一个动态数组。同时，动态数组还提供了 Copy 和 CopyRange 方法，CopyRange 方法可以复制指定范围的数据。

两个动态数组之间可以进行比较，可以直接使用“=”操作符。

使用嵌套声明动态数组可以实现多维动态数组，多维动态数组使用起来也和一般的多维数组相同，下面一段程序演示了多维动态数组的使用：

```
typedef DynamicArray<DynamicArray<AnsiString>>T2DStringArray;
void test(T2DStringArray &s_array)
{
    SetLength(s_array,10);
    for (int i=0;i<s_array.Length;i++)
    {
        SetLength(s_array[i],i+1);
        for (int j=0;j<s_array[i].Lengthj;j++)
            s_array[i][i]=itoa(i*10+j);
    }
}
```

1.3.5 流（TStream）

C++Builder 中引入了流类（TStream），并且广泛使用了它。TStream 的继承对象用于在内存、Windows 资源文件、磁盘文件和数据库字段等媒介中存储数据。

TStream 类是能在各种媒介中存储二进制数据的抽象类。VCL 定义了 TStream 抽象类的几个子类，其中三个最为重要的类是 TFileStream、THandleStream、TMemoryStream。TFileStream 类支持文件流的操作；THandleStream 支持已知文件 Windows 句柄的文件流操作；TMemoryStream 用于管理内存地址，进行内存流的操作。TMemoryStream 类非常重要，因为该类具有特殊的对象方法，支持向其他流类对象复制内容，或者从其他流类对象中读取内容，所以，TMemoryStream 类对象常常被作为流式操作中的中间对象。

TStream 中定义了两个属性：Size 和 Position。它们以字节为单位，分别表示流的大小和当前指针位置。TStream 中定义的方法用于在各种流中读、写和相互拷贝二进制数据。当然，直接创建 TStream 类的实例是没有意义的，因为该类为抽象类，它只具有虚方法而不支持实际的

数据保存功能。但可以创建 TStream 类的派生类进行实际数据处理，包括使用内存流和文件流，关于使用文件流和内存流编程的内容请参看本书第 3 章。

TStream 类的重要属性和方法

- Position 属性。

声明：__property int Position = {read=GetPosition, write=SetPosition, nodefault};

Position 属性指明流中读写的当前偏移量。

- Size 属性。

声明：__property int Size={read=GetSize,write=SetSize,nodefault};

返回以字节为单位的流对象的大小。

- CopyFrom 方法。

声明：int__fastcall CopyFrom(TStream* Source, int(Count);

该函数从 Source 参数指定的流中拷贝 Count 个字节到当前流中，并将指针从当前位置移动 Count 个字节数，函数返回值是实际拷贝的字节数。

- Read 方法。

声明：virtual int __fastcall Read(void *Buffer, int Count) = 0;

该函数从当前流中的当前位置起将 Count 个字节的内容复制到 Buffer 中，并把当前指针向后移动 Count 个字节数，函数返回值是实际读的字节数。如果返回值小于 Count，则意味着读操作在读满所需字节数前指针已经到达了流的尾部。

- Seek 方法。

声明：virtual int __fastcall Seek(int Offset, Word Origin) = 0;

Seek 方法将流的当前指针移动 Offset 个字节，字节移动的起点由 Origin 指定。如果 Offset 是负数，Seek 方法将从所描述的起点往流的头部移动。参数 Origin 有下列三个可取值：soFromBeginning、soFromCurrent、soFromEnd。

- Write 方法。

声明：virtual int __fastcall Write(const void *Buffer, int Count) = 0;

该函数将 Buffer 中的 Count 个字节写入流中，并将当前位置指针向流的尾部移动 Count 个字节，函数返回写入的字节数。

1.4 使用 C++ Builder 5 的 VCL 增强

Borland C++ Builder 5 较之它的前一版本——Borland C++ Builder 4 有很多改变，这表现在诸多方面，包括 IDE、编译器与调试器，但更多的则是开发功能和语言方面的增强。具体到 VCL 可视化组件库方面，C++ Builder5 引入了很多新的组件和技术。本节将介绍 Borland C++ Builder 5 的新特点，并重点讨论如何使用 C++ Builder 5 的一些新技术编程。

1.4.1 C++ Builder 5 的新特点

作为最新版本的 C++ RAD 开发工具，Borland C++ Builder 5 在开发环境、分布式应用系统开发、支持已有 C++资源、Web 及 Internet 应用程序开发、数据库处理等方面，表现更为出

色，具有许多特色和新特点。

1. 全新的 IDE（集成开发环境）

C++ Builder 5 集成开发环境的用户界面更加友好，更易于使用。C++ Builder 5 采用了停驻式（Docking）工具条，可以自由组合集成开发环境窗口和工具栏的排放方式。C++ Builder 5 IDE 对于方便实际编程工作的改进是更让人惊喜的。

C++ Builder 5 集成开发环境提供以下几项新的工具或功能使代码编写工作更加轻松自如，包括：

- Code Explorer。Code Explorer 是一个位于编辑器一侧的窗口，它详细的列出了源代码中所有类的信息，包括程序中所有定义的类和类的域、属性、方法等。
- 编辑器漫游。当使用者按下 Ctrl 键不放，并使用鼠标移动到编辑器代码的类、属性、函数等的下面，鼠标指向的标示会变成具有下滑线的链接文本，单击它可以直接链接到函数等的实现或声明的代码处。
- Code Completion。该技术使编译器能够自动列出 VCL 组件的可用属性和方法供程序员选择（如图 1-7 所示），而不必手工输入冗长的代码。为了激活它，只需键入一个对象的名称，如 Label1，然后键入“->”或“.”并等待片刻即可。

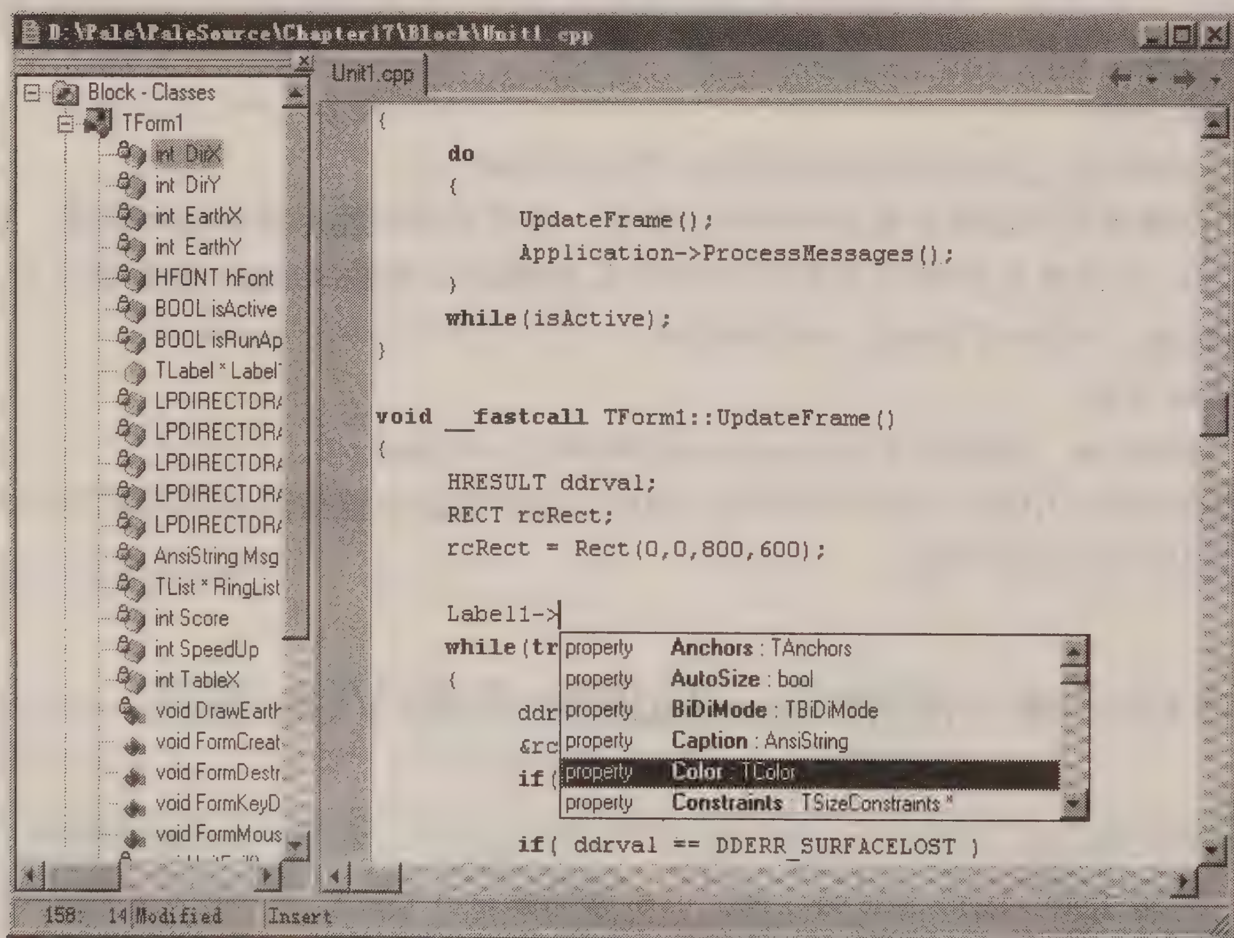


图 1-7　C++ Builder 5 的代码编辑器

- Code Template。该功能允许用户在程序中插入事先定义的代码模板。使用这项功能首先需要在编辑器中键入代码模板的标识符，而后按下 Ctrl+J 激活代码模板。
- 编辑器支持书签，新提供的编辑键映射功能可以让代码编辑器按照用户的习惯来工作。
- 新增行为列表（To-Do List）有效地管理开发计划。

- 带有树视图和数据图表视图（Data Diagram View）的 Data Module 设计器可以帮助用户充分理解程序中的数据。

Tip 可以使用代码模板实现常用表达式或函数调用的快速输入。例如，对于经常使用的 MessageBox 函数，可以为它添加一个模板。具体方法为，选择【**Tools/Environment Options**】菜单打开对话框的 Code Insight 页，在 Code templates 区中单击 Add 按钮，键入模板名（如 mb），输入说明，而后在 Code 多行文本框中输入模板代码：MessageBox (Handle,"|","",MB_OK|MB_ICONEXCLAMATION); 确定后，只要在代码编辑器中键入 mb，然后按下 Ctrl+J，就可以获得完整的模板代码文本。模板代码中竖线符说明在展开模板后，光标在编辑器上源代码的位置，如图 1-8 所示。

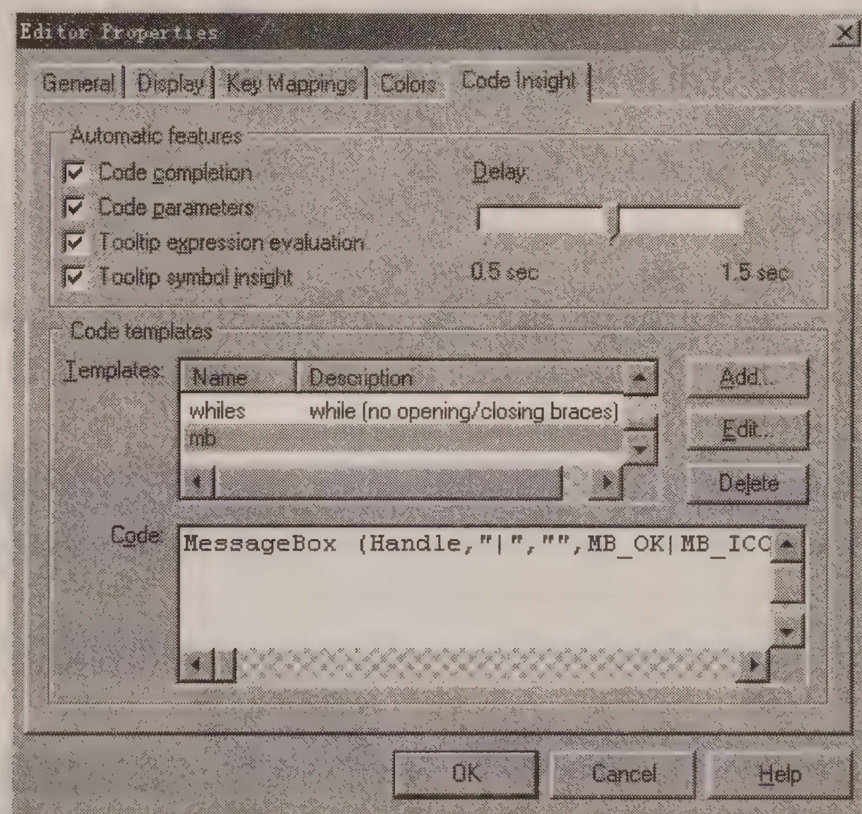


图 1-8 使用 Code templates 功能定制代码模板

2. 更强劲的编译器

C++ Builder 5 改进了对于 Borland C++OWL 和 Microsoft MFC 的兼容性。

C++ Builder 5 所使用编译器是最新的 Borland C++ 5.5 编译器，对 Borland C++ OWL 和 Microsoft MFC 提供支持。C++ Builder 5 提供的 MFC 4.2 版的函数库，可以直接编译 MSDN 与各种 SDK 中的范例程序。通过 MFC 向导，还可以生成 MFC 的代码框架。C++ Builder 附带了 vctobpru 工具，可以将 Visual C++的工程直接转换为 C++ Builder 工程。

3. 增强的 VCL 功能

在 C++ Builder 的可视化编程核心——VCL 方面，C++ Builder 5 对 VCL 进行了很多增强，提供了 PageScroller、MonthCalendar、ActionList 等新的组件，实现了 ListView、TreeView、TodBar 等组件的高级自画属性，同时原有组件有了进一步的增强。例如，TIcon 类支持多重图

标和超过 16 色的图标，Splitter 组件的 AutoSnap 功能和相应的 AutoSnap 属性，TScreen 类的 HintFont 属性和 MenuFont 属性。

另外，C++ Builder 5 提出了框架（Frames）概念，利用框架技术可以更有效地设计和重用界面；增加了 Mircosoft Office 自动化组件集，可以把 Word、Excel 和 Outlook 等 Office 程序快速集成到应用程序中；提供新的 FastMath 数学函数集（头文件：fastmath.h），FastMath 函数比 C++原有的数学函数（math.h）更快。

4. 快速开发 Web 及 Internet 应用程序

目前，基于 Internet 的开发已经具有举足轻重的地位。C++ Builder 5 在开发 Web 及 Internet 应用方面的功能进一步强化。C++ Builder 5 提供了新的 CppWebBrowser 组件以替代原有的 HTML 组件，以及 21 个 Internet 通信协议组件，用于 Internet 应用程序的开发。C++ Builder 5 同时支持 CGI、WIN－CGI、ISAPI 及 NSAPI 等标准，使开发人员利用现有的开发技术就可以用可视化的方式开发跨平台的 Web 应用程序。运用 ActiveForm/ATL 及 WebDeploy 技术，还可以实现 ActiveX 组件的 Web 分发。

5. 增强的数据库功能

C++ Builder 5 新增了 ADOExpress 组件集，提供对 Microsoft 的 ActiveX Data Objects（ADO）技术的支持。ADO 是 Mircosoft 公司建立的对各种数据库格式的高层接口（High-Level Interface），该接口已经成为访问数据库的新的标准。

C++ Builder 5 内嵌 InterBase 5.6 大型数据库的高速驱动程序，并且新增了 InterBase Express（IBX）组件集。使用 IBX 组件开发的应用程序将工作得更好，执行得更快并支持进入 InterBase 的高级特性。

6. 出色的分布式计算开发

企业向多层分布式系统跨越已经成为了一种必然趋势，目前分布式计算标准主要有 Microsoft 的 DCOM 和 OMG 的 CORBA， C++ Builder 5 对 CORBA 和 DCOM 都有极好的支持，它既适用于基于 ORB 的分布式开发，又适用于基于 COM 的 Windows 开发。C++ Builder 5 内置了 VisiBroker for C++ ORB Version 4.0，并且包含了 Event Service 和 NamingService 等标准 CORBA 服务，开发 CORBA 应用更为方便。C++ Builder 5 将 CORBA IDL 编译器集成在其开发环境中，并提供了各种向导（Wizard），可以快速生成 CORBA Client 和 Server 的源程序代码框架。

C++ Builder 5 同样为 COM 技术准备了多种向导，可以一步生成 COMOLE Automation 组件及 ActiveX 组件。C++ Builder 5 提供的 MIDAS3 同时支持所有的分布式计算标准，包括 CORBA COM 和 MTS 等，可以实现现有系统和电子商务程序之间的无缝集成。

1.4.2 使用 TActionList 组件和 TMonthCalender 组件

在 Borland C++ Builder 5 中，VCL 可视化组件库被大大加强了。就组件方面而言，变化主要表现在两方面：增加了有用的新组件和新类；原有组件和类功能的进一步增强。基本组件

方面，C++ Builder 5 增加了 ApplicationEvents 组件，以方便应用程序对象相关事件编程。另外，ActionList 是 C++Builder 4 引入的重要组件，同样是为方便通用事件的编程。这一小节中将介绍这两个组件的使用。

1. TActionList 组件

Borland 公司从 C++Builder 4 开始提出了动作项(Action)的概念。动作项是指某个用户操作命令，例如复制、剪切、粘贴、删除等等就是典型动作项。C++ Builder 5 提供了 TActionList 组件来建立和管理动作项。这些动作项可以经由这些控件的 Action 属性被链接到用户控件上（该属性为 TBasicAction 类型，并且可以在设计期间设置）。下面是关于动作项/TActionList 组件机制的具体描述：

- 动作项由一个 TAction 类定义。TAction 类型包括一些通用的属性，例如 Caption（标题）、Checked（是非状态）、Enabled（是否可执行）、HelpContext（帮助）、Hint（暗示文本）、ImageIndex（图标）、ShortCut（快捷键）等等。同时提供 Execute 方法和 OnExecute 事件，与动作项的执行相关。
- TActionList 是一个非可视组件。这个组件用于创建并管理程序中所有动作项。TActionList 组件包含了一个 ActionList 编辑器，使用 ActionList 编辑器可以创建新的动作项。另外，C++ Builder 已经预定义了一些标准动作项，例如 TEditCopy、TEditCut、TWindowClose 等等。在 ActionList 编辑器的弹出菜单中选择 New Standard Action 项，可以加入这些标准动作项。如图 1-9 所示。

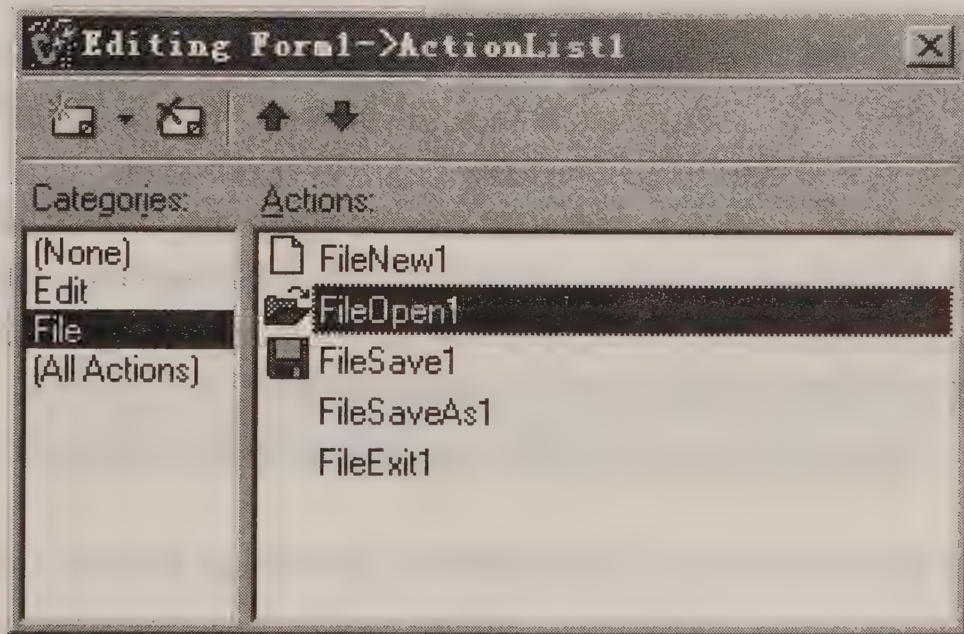


图 1-9　ActionList 编辑器的使用

- 动作项可以用来代表一个菜单项（TMenuItem）或一个按钮（TToolButton, TSpeedButton, TMenuItem, TButton, TCheckBox, TRadioButton 等等）。如果在程序中定义了一个完整的动作，就可以设置这些组件的 Action 属性为已定义动作项。这样，这些组件就会具备定义动作项的相应属性（例如，菜单项会直接获得 Caption、图标、Checked、Enabled、ShortCut 等等属性，假设程序中已经设置了 Action 的对应属性）。同时，动作项的 OnExecute 将等价于这些组件的 OnClick。
- 动作项还常常被用来作为组件特定控制，例如在 TRichEdit 组件、TMemo 组件的

Copy、Cut、Paste 动作。在我们自定制 VCL 组件时，可以通过继承 TAction 类定义组件的动作项。

下图显示了 Action 对象、ActionList 对象和动作的实现者 SpeedButton 对象之间的关系。ActionList1 组件包含了一个名为 Copy1 的动作项，Copy1 动作的实现者是一个 SpeedButton。通过 SpeedButton1 的 Action 属性链接 Copy1 动作项，而 Memo1 组件是动作目标——Copy1 动作完成将 Memo1 组件选中的文本复制到剪贴板，当用户单击 SpeedButton1 后将实际执行这个动作，如图 1-10 所示。

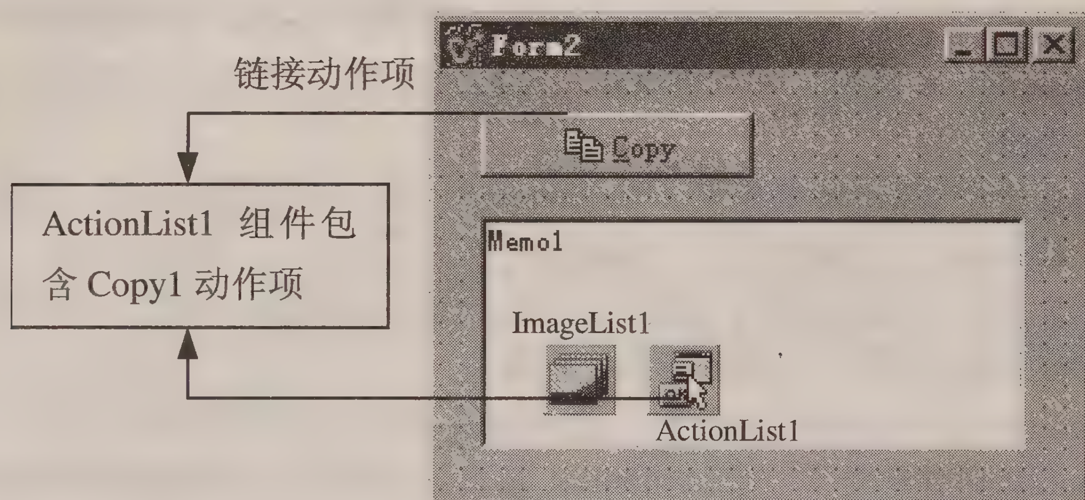


图 1-10　Action 的实现机制

2. TMonthCalender 组件

使用 TApplicationEvents 可以方便地截获 Application 全局对象的所有事件。当放置 TApplicationEvents 组件在某个 Form 上，通过 TApplicationEvents 组件的事件，就可以处理 Application 全局对象的同名事件。

任何一个 Form 对象都可以拥有一个自己的 TApplicationEvents 组件，当某个 Application 对象的事件发生时，所有的 TApplicationEvents 组件的同名事件都将被触发。通过 Activate 方法，可以改变多个 TApplicationEvents 组件间接受事件的顺序。当某个 TApplicationEvents 组件执行了 Activate 方法，将使该组件首先接受到某个 Application 事件。

为了避免某个 TApplicationEvents 组件接收 Application 事件，可以调用 CancelDispatch 方法。

以下将创建一个简单的范例程序，该程序演示了 TActionList 组件和 TMonthCalender 组件的使用。

在程序窗体中加入一个 ActionList 组件，通过 ActionList 建立了一个简单的名为 Action 的动作项，同时包含了一个 TApplicationEvents 组件，通过该组件我们处理了所有类型的 Applicaiton 事件，包括 OnActionExecute（动作项执行）、OnActionUpdate（动作项执行 Update 方法）、OnActivate（应用程序激活）、OnDeactivate（应用程序失去焦点）、OnException（当异常发生）、OnHelp（执行上下文帮助）、OnHint（当鼠标移动到新的可能产生暗示文本的组件发生）、OnIdle（应用程序空闲时发生）、OnMessage（收到某个 Windows 消息）、OnMinimize（最小化时发生）、OnRestore（由最小化恢复窗口时发生）、OnShortCut（用户按下应用程序定义的快捷键）、OnShowHint（暗示文本框显示）。最后把事件的发生在列表框中记录下来。

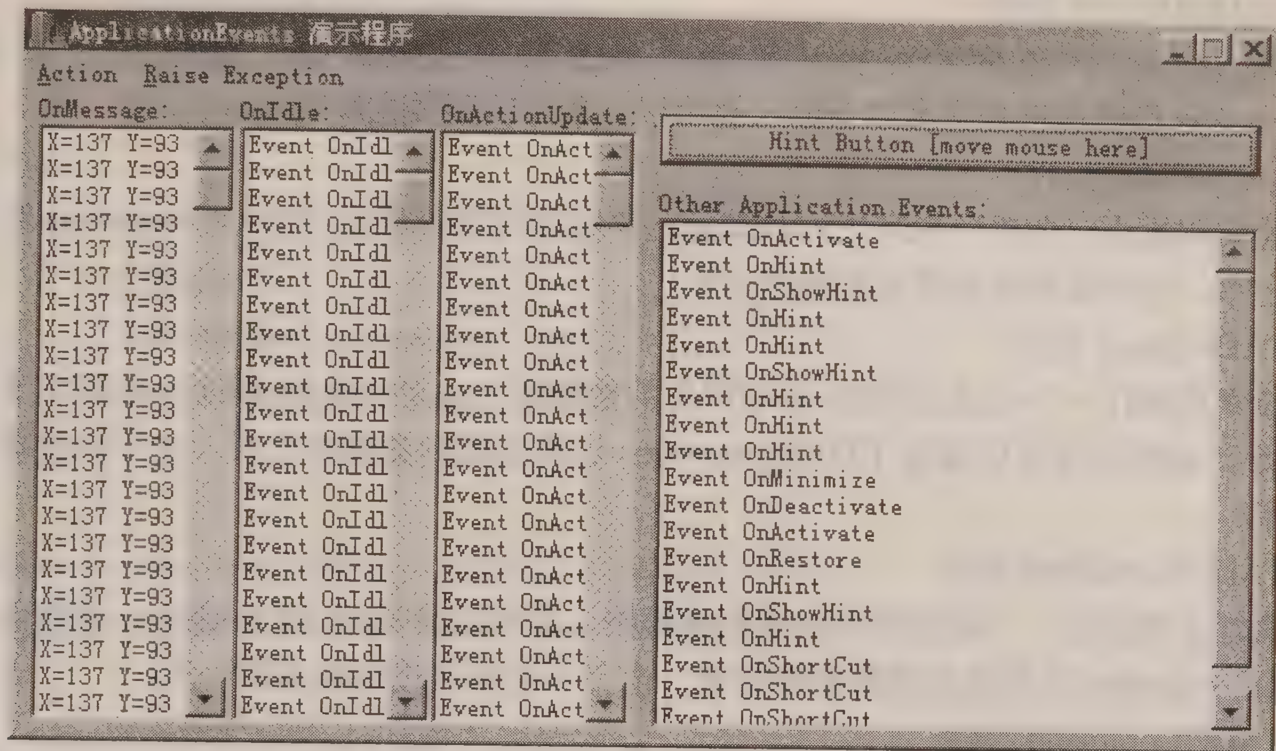


图 1-11　TAppicationEvents 和 TActionList 组件的示例程序

1.4.3　ADOExpress 组件编程

ADO（ActiveX Data Objects）是微软提供对各种数据库格式的高层接口。使用这种接口的数据库又称为 OLE DB 数据库。OLE DB 数据库可以使我们方便地访问各种类型的数据库，包括关系型或非关系型数据库、E-Mail 和文件系统、文本和图形以及各种自定义商用对象。

C++ Builder 5 提供了一系列组件用于基于 ADO 的数据库程序设计，这一系列组件被称为 ADOExpress。C++ Builder 组件选项板的 ADO 页包含了这组组件，见图 1-12。这些组件提供应用程序进行连接到 ADO 数据仓库、执行命令、检索数据库中的数据等操作。

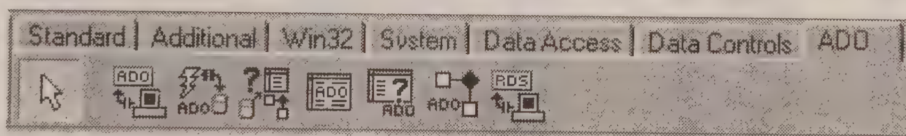


图 1-12　ADOExpress 组件组

ADO 连接和数据集组件中的大多数都分别与 BDE 连接和数据集组件中的一个类似。例如，TADOConnection 组件功能上类似于 BDE 应用程序中的 TDatabase 组件，TADOTable 组件类似于 TTable 组件，TADOQuery 组件类似于 TQuery 组件，TADOStoredProc 组件等价于 TStoredProc 组件。在使用上，ADO 组件也类似于 BDE 中的同类组件。TADODataSet 组件在 BDE 中没有直接的对应者，但提供了许多与 TTable 和 TQuery 组件同样的功能；TADOCommand 在 BDE 中也没有直接的对应者，其提供的功能是 ADO 编程中专有的。

下面对所有 ADO 组件作一概括的介绍。

- TADOConnection 组件

用于建立与 ADO 数据库的连接。ADO 数据集和命令组件可以共享这个连接，以执行命令、检索数据以及对源数据进行操作。

- TADODataSet 组件

用于检索和操作数据。可以从单个或多个表中检索数据。该组件既可以通过 TADOConnection 组件连接到数据库仓库，也可以直接连接到数据库仓库。

- TADOTabel 组件

用于检索和操作一个单一表产生的数据集。该组件既可以通过 TADOConnection 组件连接到数据库仓库，也可以直接连接到数据库仓库。

- TADOQuery 组件

用于检索和操作一个合法的 SQL 语句产生的数据集，也可以执行 DDL（数据定义语言）的 SQL 命令。该组件既可以通过 TADOConnection 组件连接到数据库仓库，也可以直接连接到数据库仓库。

- TADOStoredProc 组件

用于执行存储过程。存储过程可以是检索数据，也可以是执行 DDL 命令。该组件既可以通过 TADOConnection 组件连接到数据库仓库，也可以直接连接到数据库仓库。

- TADOCommand 组件

主要用于执行命令（即不需要返回结果的 SQL 语句）。也可以是一个数据集组件一起使用来从一个表中检索数据。

下面的示例程序演示了使用 ADOExpress 组件打开、显示和编辑一个 Access 数据库。见图 1-13。该程序用到了 ADO 组件 ADOConnect 和 ADODataSet 以及原有数据库组件 DataSource 和 DBGrid。

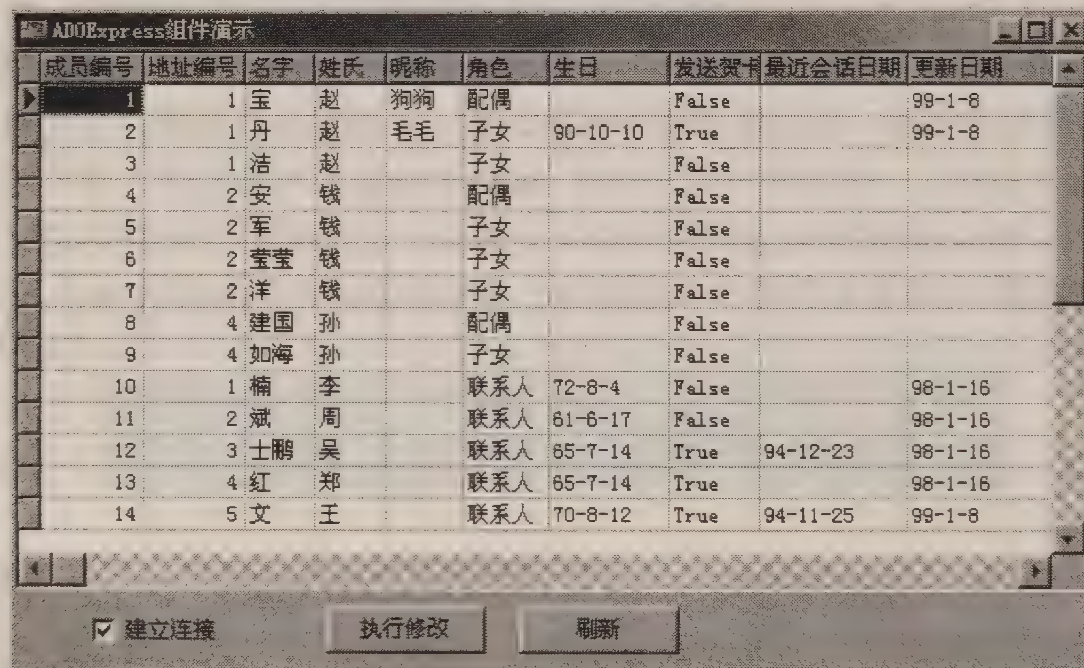


图 1-13　ADOExpress 组件的示例程序

Source　使用 ADOExpress 组件编写数据库程序

```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
        if(CheckBox1->Checked)
        {   //建立连接状态
```

```
            ADOConnection1->Open("", "");
            ADODataSet1->Connection=ADOConnection1;
            ADODataSet1->Open();
        }
        else
        {  //关闭连接
            ADODataSet1->Connection = NULL;
            ADOConnection1->Close();
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        CheckBox1->Checked = true;
        ADODataSet1->Close();
        ADODataSet1->CommandType=cmdTable;
        ADODataSet1->CommandText="家庭成员";
        ADODataSet1->Open();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCloseQuery(TObject *Sender, bool &CanClose)
{
        if(ADODataSet1->Active)
        {
                try{
                        if (ADOConnection1->Connected)
                                UpdateData();
                }
                catch(Exception &E) {
                        Application->HandleException(Sender);
                        CanClose = MessageDlg("数据改变并且没有存盘，确实退出吗？",mtConfirmation,
                        TMsgDlgButtons() <<mbYes <<mbNo <<mbCancel, 0)== mbYes;
                }
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
        ADOConnection1->ConnectionString="FILE NAME="
            +ExtractFilePath(ParamStr(0))+"DATA.UDL";
```

```
        CheckBox1->Checked = true;
        ADODataSet1->Open();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::UpdateData()
{
        CheckBox1->Checked = true;
        ADODataSet1->UpdateBatch(arAll);
}
//---------------------------------------------------------------------------
```

程序中“DATA.UDL”是一个预先建立的数据连接文件，通过该文件指向实际的数据库文件——ADDRBOOK.mdb。区别于 BDE，在 ADO 数据库编程中，通过 UDL 文件与数据库建立连接是常用的方式。

1.4.4 框架（Frame）技术编程

框架（Frame）是 C++ Builder 提供的一种代码复用技术，是 C++ Builder 5 新增加的功能之一。

框架类似于窗体，也是一个其他组件的容器。与窗体一样，框架可以自动创建和释放其上的组件，并在组件间建立父子关系。不同的是，框架是一种定制的组件，框架可以保存在组件选项板上，可以向其他组件那样，放置在窗体、其他框架或其他容器对象上。在一个框架创建并保存后，它像窗体一样作为一个单元来发挥作用。当我们修改了某个框架，使用该框架的窗体或其他框架会随着改变。

为了建立新的框架，选择【**File/New Frame**】菜单命令（或选择【**File/New**】命令后从图标中选择），然后将需要的组件或其他框架放置在新建立的框架上。在设计期间，通过选择【**View/Forms...**】命令，可以选择并显示当前工程中包含的任何框架。

框架可以保存在组件选项板上，作为组件模板。要将框架放置在组件板上，可以在窗体设计窗口打开框架，在框架上使用弹出菜单中的 Add to Palette。打开 Component Template Information 对话框，在对话框中为框架选择一个名字、所在组件页和图标。

在应用程序中使用框架，只需要将框架放置在窗体上即可。窗体设计窗口提供了两种方法将框架加入到应用程序中：

- 从组件板选择预定义作为组件模板的框架，并放置到窗体等可以容纳框架的容器上。如果需要，C++ Builder 会提示当前工程中是否可以包含框架。
- 从组件板的 Standard 页选择框架，然后单击需要放置框架的容器，这时将打开框架选择对话框列出当前工程中包含的所有框架，可任选一个加入。

当将框架放置到窗体或其他容器对象上时，C++ Builder 将定义一个由所选框架派生的新类。这就意味着对源框架的修改，将会传给被嵌入的框架。但对被嵌入的框架的修改，不会反映到源框架。

可以将框架加入对象库中以实现框架共享。方法是，打开任何一个包含框架的工程，在

窗体设计窗口的弹出菜单中选择 Add to Repository。

下面的示例程序演示了框架技术的使用，该程序包含了三个框架：FancyFrame 框架包含 DBMemo 组件和 DBImage 组件；DataFrame 框架嵌入了一个 FancyFrame 框架，并包含了其他一些数据库组件；MasterDetailFrame 框架则是嵌入了两个 DataFrame 框架，如图 1-14。

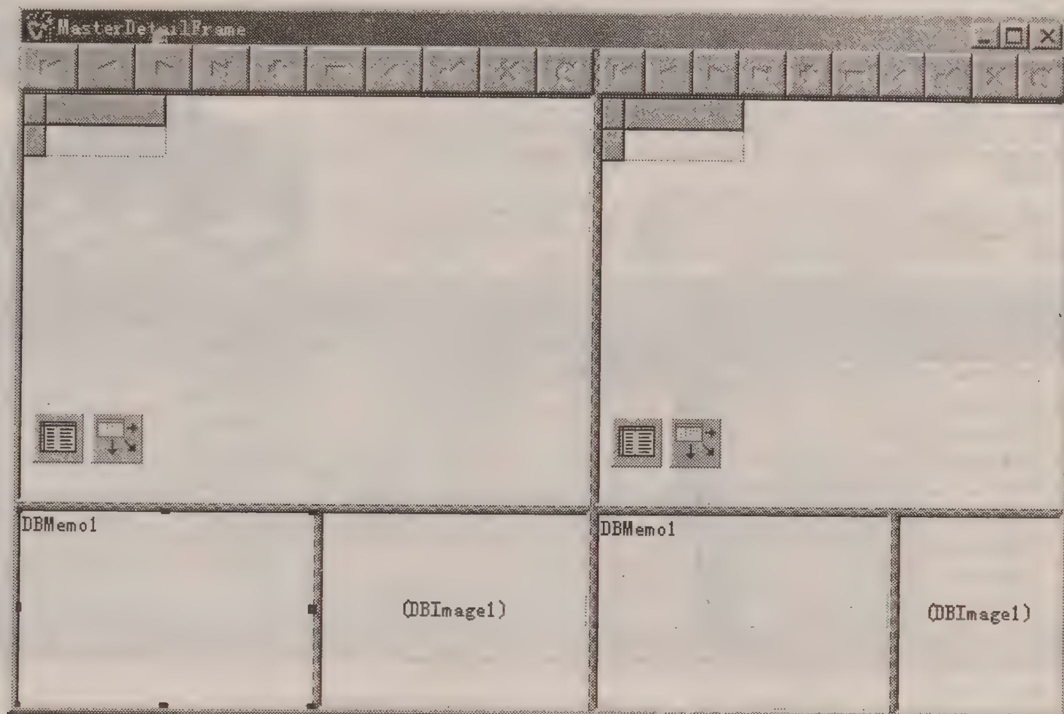


图 1-14　由两个 DataFrame 框架建立的 MasterDetailFrame 框架

在应用程序窗体中包含一个 DataFrame 框架和一个 MasterDetailFrame 框架。该程序的关键代码如下所示：

```
SimpleFrame->Table1->TableName="Biolife.db";
SimpleFrame->FancyFrame1->DBMemo1->DataSource=SimpleFrame->DataSource1;
SimpleFrame->FancyFrame1->DBMemo1->DataField="Notes";
SimpleFrame->FancyFrame1->DBImage1->DataSource=SimpleFrame->DataSource1;
SimpleFrame->FancyFrame1->DBImage1->DataField="Graphic";
SimpleFrame->Table1->Open();
delete MDFrame->MasterFrame->FancyFrame1;
delete MDFrame->MasterFrame->Splitter1;
MDFrame->MasterFrame->Table1->TableName="Customer";
MDFrame->MasterFrame->Table1->Open();
delete MDFrame->DetailFrame->FancyFrame1;
delete MDFrame->DetailFrame->Splitter1;
MDFrame->DetailFrame->Table1->MasterSource=MDFrame->MasterFrame->DataSource1;
MDFrame->DetailFrame->Table1->MasterFields="CustNo";
MDFrame->DetailFrame->Table1->IndexName="CustNo";
MDFrame->DetailFrame->Table1->TableName="Orders";
MDFrame->DetailFrame->Table1->Open();
```

注意，程序中使用 delete 操作符销毁框架上的组件对象达到了修改界面的目的，程序运行时的效果如图 1-15 所示。

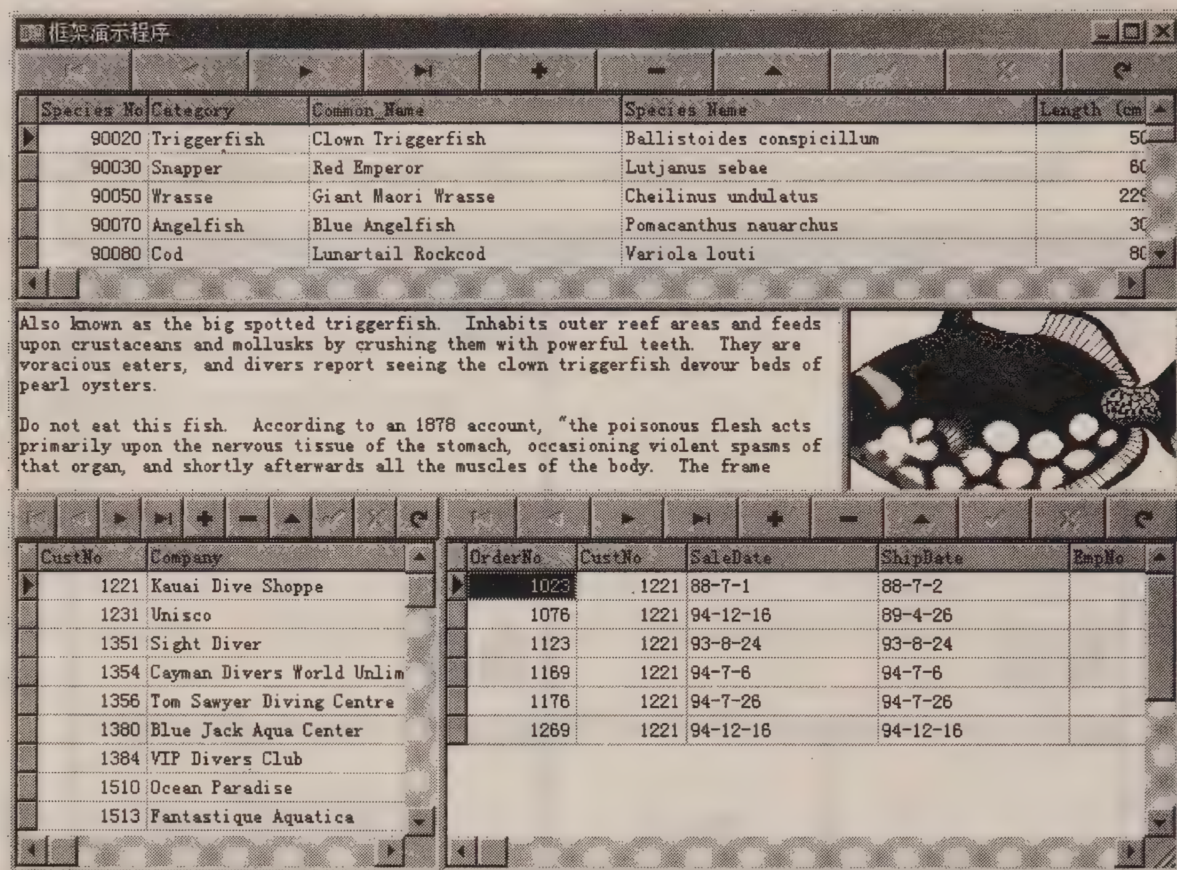


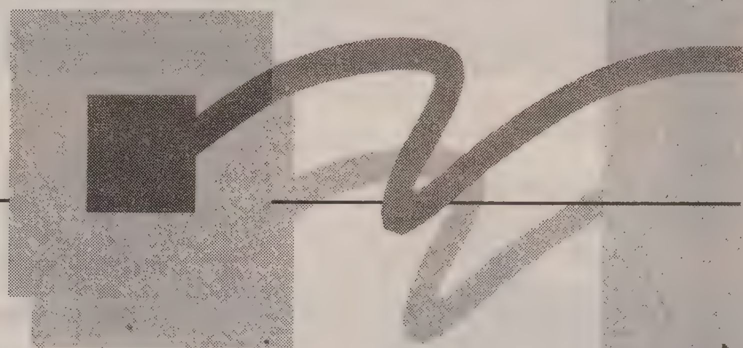
图 1-15　使用框架技术的示例程序

第2章

功能齐全的多文档书写器

——高级文本处理

文本编辑组件的高级用法
多文档界面（MDI）技术
字体和格式控制
剪贴板功能及查找、替换功能
高级文件编辑器功能



本章导读

Windows 95/98 的写字板应用程序为人们所熟知。实际上利用 C++ Builder 的强大功能，开发相同甚至更好的字处理程序也并非难事。本章我们将编程实现一个比写字板更好的字处理应用程序——BCB 书写器。该程序支持纯文本格式和 RTF 格式，实现了 Windows 95/98 下写字板（WordPad）程序 90%以上的功能和一些 WordPad 没有提供的高级功能。

本章通过创建 BCB 书写器程序，讲述关于文本编辑处理方面的编程问题以及多文档界面（MDI）技术。当然，对于 BCB 书写器范例程序中高级功能的实现，涉及其他编程技术是不可避免的，而这些内容可能才是本章真正的精华所在。

本章内容包括:

- Borland C++ Builder 提供的各种文本编辑组件的高级用法。
- 多文档界面（MDI）和多页面界面（MPI）技术。
- 实现范例——BCB 书写器程序，介绍各种功能的处理及实现技巧，包括：完整的字体/格式控制；文本编辑功能（复制、剪切、粘贴等）；查找、替换文本功能和文件的创建、打开、保存和打印。
- 深入的内容。包括通过消息发送获得当前光标所在位置；打开历史文件列表的实现；以及多文档界面父窗体背景贴图的绘制。

2.1 文本编辑组件的高级用法

输入与输出是应用程序的基本功能，也是应用程序与用户交互的必然方式。尽管在理论上程序员可以通过 Windows 键盘消息响应来直接处理键盘输入，但没有人愿意这样做，因为 Win32 操作系统已经提供三种基本风格的文本编辑对象：单行编辑框、多行编辑框和多文本编辑框（对应于 C++ Builder 中 TEdit、TMemo 和 TRichEdit 三个基本输入输出对象），可用于获得字符串输入和进行字符串输出，以至建立简单的文本编辑器。C++ Builder 提供多种不同的文本编辑组件，它们都继承于 TCustomEdit 对象，从简单到复杂，代表不同方面的应用。

另外，C++ Builder 还提供 TMaskEdit 组件，该组件为 TEdit 组件的增强版本，提供可以定制掩码的输入框。

2.1.1 TEdit 组件和 TMaskEdit 组件

1. 单行编辑框组件 TEdit

TEdit 组件是一个应用最为广泛的组件，大多数可视化应用程序都会用到。Text 属性是它的基本属性，通过读写 Text 属性就可完成接受用户输入，也可以通过 Text 属性显示单行文本，但这种情况下更多则是使用 TLable 组件或 TStaticText 组件。

TEdit 组件使用起来非常简单，但我们往往希望能够对用户输入进行特殊的控制。TEdit 组件能够影响文本编辑框输入条件的属性包括 AutoSelect、CharCase、MaxLength、PasswordChar 等。

- AutoSelect 属性。

自动选择。用来设置当 TEdit 组件成为用户焦点时，编辑框中的文本是否被自动选择。应用 AutoSelect 属性的明显例子是 C++ Builder 中的对象编辑器（Object Inspector），其中所有编辑框的 AutoSelect 属性均被置为 true。

- CharCase 属性。

用来控制编辑框内字符的大小写，可选的值有 ecNormal、ecUpperCase、ecLowerCase。

- MaxLength 属性。

控制文本编辑框接收字符的数量，置为 0 时表示不受限制。

- PasswordChar 属性。

用于指定密码输入时的掩盖字符。例如可以设置该属性为“*”，无论用户输入什么字符，在编辑框内都以“*”显示。此属性为缺省值 0 时，表示为非密码输入状态。

但更多实际情况中需要建立用户输入条件限制用户输入，例如只想让用户在编辑框中输入数字，而不需要接受字母或其他字符。可以处理 TEdit 组件的 OnKeyPress 事件，在该事件响应函数中分析用户输入字符是否是合法的，如果不是，则将键值变为空字符，这样文本编

辑组件将不会处理它。下面是示例代码：

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, char &Key)
{
    // 检查用户输入字符是否合法，其中 8 为 BackSpace 键值
    if ((Key>='0'&&Key<='9')||Key==VK_LEFT||Key==VK_RIGHT||
        Key==VK_DELETE||Key==8)
        return;
    else {
        Key = 0;
        MessageBeep(MB_OK);
    }
}
```

2. TMaskEdit 组件

为了进一步定制用户输入，可以考虑使用 C++ Builder 的 TMaskEdit 组件，这个组件用于实现非常复杂的输入控制。TMaskEdit 组件的核心属性称为 EditMask，它说明每个用户输入字符的性质——是否为大写、小写或数字，并可指定相似条件的字符串。

C++ Builder 为 EditMask 属性提供了属性编辑器（如图 2-1 所示），这个编辑器允许用户编辑输入掩码（Input Mask），同时它要求用户为输入提供占位符字符（Character for Blanks），并决定与结果字符串一起保存出现在掩码中的字母（Save Literal Characters）。一个完整的掩码包括三个域，这三个域以分号隔开：第一个域是掩码本身；第二个域用于指定掩码字面上的特性是否作为数据的一部分被保存；第三个域指定占位符，在应该由用户输入字符的地方，但还未输入时显示该字符。

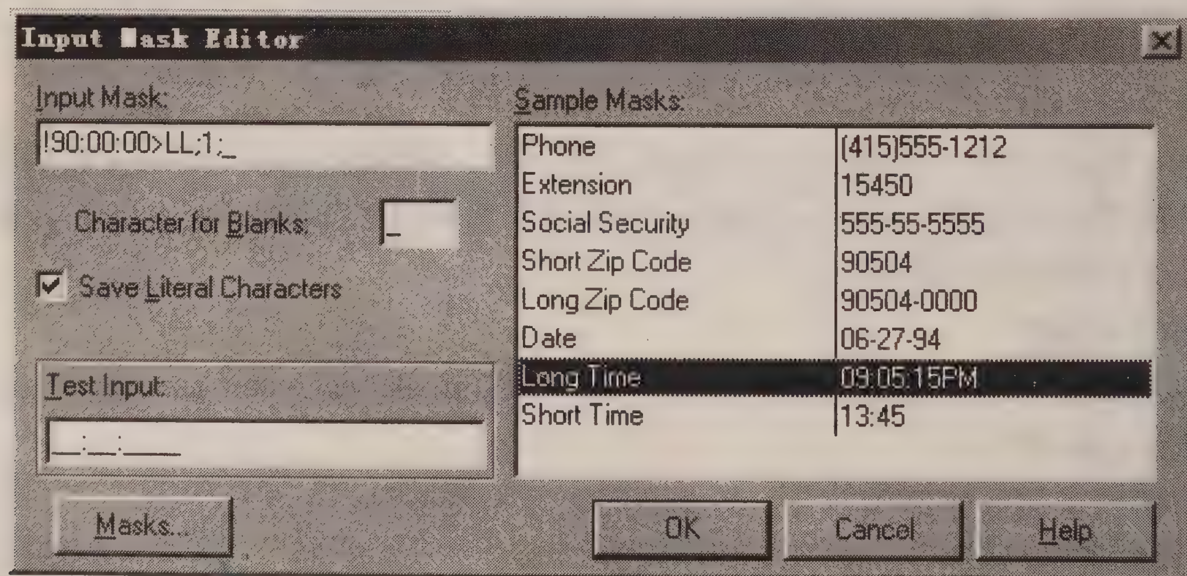


图 2-1　TMaskEdit 组件的 EditMask 编辑器

例如规定了一些特定意义的字符来定制用户输入，常用的特殊标志字符包括：

>　　在其后的输入全部用大写字母表示直到结束或者遭遇<字符。

<　　在其后的输入全部用大写字母表示直到结束或者遭遇>字符。

\	用来区分一般字符和掩码使用的特殊字符。例如“\L”表示大写字母 L，而不是掩码标志字符 L。
L	在此出接受英文字母。
0	在此出接受数字。

另外，单击 MaskEditor 的 Masks 按钮，可以从文件导入预定义的输入代码。



TMaskEdit 组件也支持多字节语言的掩码输入，例如可以定制汉字掩码。对于双字掩码，只需要将各种用于字母输入的标志符，包括 L、l、A、a、C 和 c 双写即可，例如 LL 就可以支持一个汉字的输入。

2.1.2 TMemo 组件和 TRichEdit 组件

1. 多行文本编辑组件 TMemo

TMemo 组件可用于输出和显示较多的文本。TMemo 组件可以管理多行文本，但 TMemo 组件保留了 Windows 3.X 的某些限制：每一行最多可以有 255 个字符，其最大数据量为 32KB。

完全可以利用 TMemo 组件来编制一个类似 Windows 记事本的程序，甚至可以把 TMemo 组件看作是一个可受控制记事本，所有 Windows 记事本的功能 TMemo 组件都有所支持，甚至包含了实现剪贴板功能及撤销、全选功能的弹出菜单——如果不设置 TMemo 组件的 PopupMenu 属性，具有上述功能的弹出菜单将被自动实现。

TMemo 组件的基本属性是 Lines 属性，它用于存放 TMemo 组件中的文本，该属性为 TStrings 类。TStrings 类是 TStringList 类的抽象父类，用于存放一个字符串列表，其基本属性是 Strings，这个属性这样定义：

```
_property System::AnsiString Strings[int Index] = {read=Get, write=Put};
```

所以，当需要更改 TMemo 编辑框的第 6 行文本时，程序代码为：

```
Memo1->Lines[5]="C++ Builder 5 is the BEST!";
```

另外，TStrings 类继承了 TList 链表类的 Add、Delete 等方法用于控制其内容。

TMemo 组件也提供一些用户输入控制功能，包含于以下几个重要属性。

- WordWrap 属性。

此属性用于设置是否自动折行。自动折行时水平卷轴失效。

- WantReturns/WantTab 属性。

决定 TMemo 编辑框是否接受回车键或 Tab 键。

- SelLength/SelStart/SelText 属性。

SelStart 属性表示在 TMemo 编辑框中被选择的文本的起始点：SelLength 表示 TMemo 编辑框中被选择的文本的长度；SelText 属性表示被选择的文本字符串。这三个属性确定了 TMemo 编辑框被选文本的信息。

此外，TMemo 组件提供了整套对应于 Windows 记事本程序功能的方法（这些方法实际上继承自 TCustomEdit 类，其他文本编辑组件同样具备），包括：

Clear	清除全部文本。
ClearSelection	清除编辑框中被选择的文本。
CopyToClipboard	将编辑框中的被选文本拷贝到剪贴板。
CutToClipboard	将编辑框中的被选文本剪切到剪贴板。
PasteFromClipboard	将剪贴板中的文本粘贴到编辑框中 SelStart 位置。
SelectAll	全选编辑框中的文本。
Undo	撤销上一次的动作（仅支持一级）。

2. 多格式文本编辑组件 TRichEdit

如果需要提供处理大量文本（多于 32KB）的能力或需要文本字体、格式控制等功能，则可以使用 TRichEdit 组件。TRichEdit 组件与 TMemo 组件类似，但可以用于编辑和处理 RTF（Rich Text Format）格式的文本，并提供了与之对应的属性和方法。

TRichEdit 组件包括了与 TMemo 组件相同的属性，如 SelLength，SelStart，SelText 等。其他重要属性有：

- Lines 属性。

与 TMemo 组件类似，用于纪录编辑框中的纯文本的内容。

- DefAttributes 和 SelAttributes 属性。

声明：__property TTextAttributes* DefAttributes = {read=FDefAttributes, write=SetDefAttributes};

两属性均用于描述文本的字体属性，包括字体名称、大小、颜色等。DefAttributes 属性用于新增文本，而 SelAttributes 属性用于被选择的文本，这两属性为 TTextAttributes 类型。对于 TTextAttributes 类，它有一个 Assign（赋值）方法是非常重要的。通过它可以把一个 Font 类的值赋给 TTextAttributes 类。

- Paragraph 属性。

声明：__property TParaAttributes* Paragraph = {read=FParagraph};

此属性用于设置或返回当前所在段落的编排格式。

- PlainText 属性。

此属性设为 true 时，编辑框内的内容将作为纯文本处理，否则视为 RTF 处理。

本章主要范例程序是一个基于 TRichEdit 组件的文本编辑器程序。以上仅仅是对 TRichEdit 组件的一个非常简单的介绍，相信通过 BCB 书写器范例程序的实现，读者将能够更加深刻地了解和掌握 TRichEdit 组件。


RTF（Rich Text Format）格式是 Mircosoft 公司专为 Windows 设计的文本格式，它在纯文本的基础上加入了文字字体属性，段落特性等功能。该格式和 Mircosoft Word 的文档格式较为类似，某种意义上可以将 Mircosoft Word 文档视为 RTF 格式的扩展。另外，在制作帮助时 RTF 格式是一个必须的中间格式（参看第 8 章）。

2.2 多文档界面（MDI）和多页面界面（MPI）技术

多文档界面（MDI，Multiple Documant Interface）是能够同时处理多个文档的应用程序用户界面和窗体结构。MDI 是多窗体结构，包括唯一的父窗体和多个子窗体。父窗体即是应用程序的主窗体，它像容器一样包含众多对应文档的子窗体。主窗体与子窗体之间是隶属关系，而子窗体之间是平等关系。

在 MDI 程序中，每个子窗体都是独立的窗体，尽管许多子窗体可以被同时显示，但同一时刻只有一个子窗体被激活，并且子窗体被主窗体限制在某个区域内。

2.2.1 多文档接口与 MDI 应用程序

多文档接口标准原是 IBM 设立的公共用户存取（CUA，Common User Access）标准集的一个组成部分。时至今日，它已成为了 Windows 应用程序的标准用户接口的规范，大量的应用都使用了这个标准用户接口。

在不存在可视化开发工具的时代，用 Windows API 写 MDI 应用程序使大多数程序员望而怯步。这是因为编写一个 MDI 的应用至少要完成以下步骤：

(1) 注册父窗体（主窗体）和子窗体类，而 MDI 子窗体类的结构和一般的非 MDI 应用程序的子窗体的类结构不同。

(2) 创建主窗口和子窗口。

(3) 写主消息循环，也和一般的非 MDI 应用不一样。

(4) 写主窗口函数，回调函数。

(5) 创建子窗口。

后面的几步都要处理大量特殊的消息，此后才能真正开始编写在 MDI 应用程序中需要实现的功能。不仅如此，每一个功能模块的加入仍要考虑 MDI 界面程序的一些特殊因素。可以说，从底层创建 MDI 应用程序是相当艰难的。

然而如果应用 C++ Builder，这样的噩梦将没有机会再次出现。C++ Builder 中实现 MDI 应用只需在窗体设计期间将窗体的 FormStyle 属性进行改动——主窗口的 FormStyle 设为 fsMDIForm，子窗口的 FormStyle 设为 fsMDIChild。而后开发人员所要做的就是在程序中使用 new 命令生成新的子窗体类的实例，一个新的子窗体也就随之生成了。例如：

```
TChildFrom *ChildForm=new TChildForm();
```

Borland C++ Builder 中甚至存在更加简单的方法：什么也不用做，直接利用 C++ Builder 提供的 MDI 应用程序的模板。在菜单中选【File/Projects】，然后在 New Items 对话框中选择 MDI Application，C++ Builder 将自动生成一个完整的 MDI 应用程序框架，如图 2-2 所示。

对于多文档界面 MDI 技术，还涉及菜单融合，子窗口排列等问题。关于这些问题和多文档界面的具体编程细节，将在本章实现 BCB 书写器范例程序时进行具体讨论。

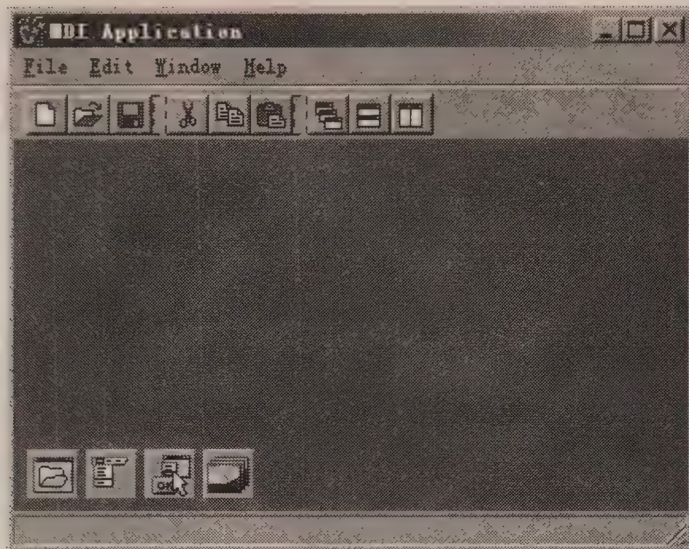


图 2-2　C++ Builder 生成的 MDI 应用程序框架

2.2.2　多页面界面 MPI

多页面界面（MPI，Mutilple Page Interface）是来自 Windows 3.X 的一种应用程序界面形式。多页面界面非常友好，它由一个窗体和多个页面组成， 关于每个页面的信息列在窗体底部的标签（Tabs）上，用户可通过选择标签来进行页面切换。 每次只有一个页面显示在窗体中。MPI 较 MDI 使用更为方便，且切换速度更快。Borland C++ Builder 4 中集成开发环境中的代码编辑（Code Editor）窗体就是 MPI 应用在文本编辑中的实例。

实现多页面界面的应用在 C++ Builder 中同样非常简单。C++ Builder 提供一个现成的组件 TTabSet。TTabSet 组件的关键属性是 Tabs 属性。Tabs 属性是一个 TStrings 类型的值。Tabs 属性的每一个字符串，对应着 TTabSet 可视组件的一个页标。通过设置 Tabs 属性，就可以创建和显示不同的页。

另一个重要的属性是 TabIndex，这个属性用于描述当前页的序号。

我们可以利用多页面技术创建一个多页记事本。这个记事本包含着五个不同的页面，用户可方便的在不同页面间切换（如图 2-3 所示）。

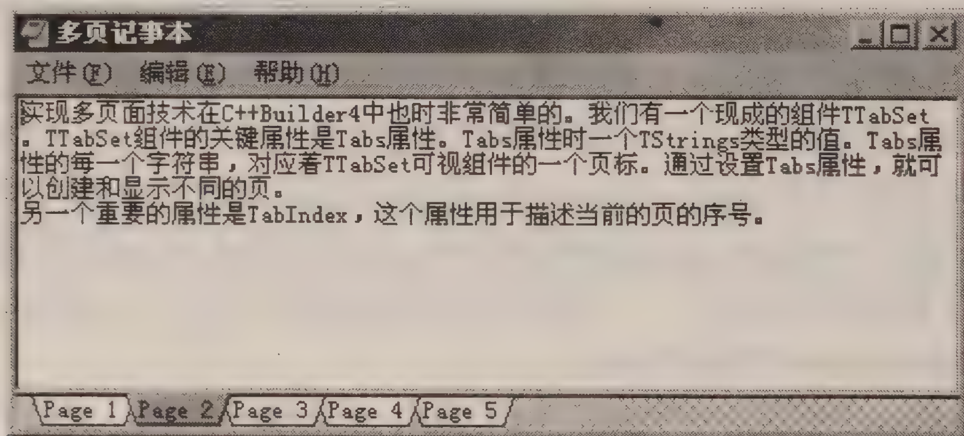


图 2-3　MPI 示例：多页记事本

MPI 应用程序界面与 MDI 界面有本质的不同。MPI 界面仅仅有一个窗口，只是被分成多页显示，并且这是一种模拟的分页。实际上，在程序用于编辑的 Memo 编辑框仅存在一个，

被切换的仅是 TTabSet 组件上的标签。

在“多页记事本”范例中，定义了名为 Pages 的 TStringList 类型的数组：

```
TStringList *pages[5]; // 包含每页文本内容数组
```

此数组用于分别存储五个页的内容，在切换时即把存储的内容写入 Memo 编辑框，从而实现了多页编辑的功能。通过下面的代码段可以明显地看到这一点。

Source 多页记事本程序的关键代码

```
//---------------------------------------------------------------------------
void __fastcall TForm1::TabSet1Click(TObject *Sender)
{
        // 页面标签单击的响应函数
        Memo1->Text = pages[TabSet1->TabIndex]->Text;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::TabSet1Change(TObject *Sender, int NewTab,
    bool &AllowChange)
{
        // 页面标签被改变时的响应函数，注意此时 TabIndex 属性还未改变
        pages[TabSet1->TabIndex]->Text = Memo1->Text;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Open1Click(TObject *Sender)
{
        // 文件打开的处理方法。其中 FileNames 是程序中定义的字符串数组
        if (OpenDialog1->Execute()){
                Memo1->Lines->LoadFromFile(OpenDialog1->FileName);
                FileNames[TabSet1->TabIndex]=OpenDialog1->FileName;
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Save1Click(TObject *Sender)
{
        // 文件存档的处理。如果 FileNames 为空，则进行 SaveAs
        if (FileNames[TabSet1->TabIndex]==""){
                if (SaveDialog1->Execute()){
                        Memo1->Lines->SaveToFile(SaveDialog1->FileName);
                }
        }
        else Memo1->Lines->SaveToFile(FileNames[TabSet1->TabIndex]);
```

```
}
//---------------------------------------------------------------------------
```

以上例子是一个 Windows 3.X 风格的多页组件的应用，而其实还有另外三种多页风格的组件，分别是 Win32 风格的 TTabControl 组件和 TPageControl 组件、Windows 3.X 风格的 TTabbedNoteBook 组件。四个多页组件都是从 TCustomTabControl 继承而来，多页风格组件的优点在于可以在有限的空间中尽量多地存放信息，而且便于把信息分类。这类组件的使用和处理方法是差不多的。

Win32 风格的 TTabControl 组件和前面用到的 TTabSet 组件极为相似，该组件支持为窗体添加页面标签供用户选择。TTabControl 和 TTabSet 组件只用在外观不变（即每页上显示的组件不变）而内容变化的多页应用中使用。

TPageControl 组件和 TTabbedNoteBook 组件类似。这两个组件的功能更加强大，它们提供真正意义上的不同页面，也就是说，在每一页中可以放置不同的组件。对于 TPageControl 来说，它的每一页都是一个 TTabSheet 组件，这个组件是一个容器，开发人员可以在上面放置不同的组件，每个 TabSheet 具有自己的属性。

Win32 风格的多页组件（TTabControl 和 TPageControl 组件）提供一些界面方面的新特性。例如，支持为页面标签提供图标标识（如图 2-4 所示），页面标签可选 tsTabs、tsButtons、tsFlatButtons 三种不同风格等。

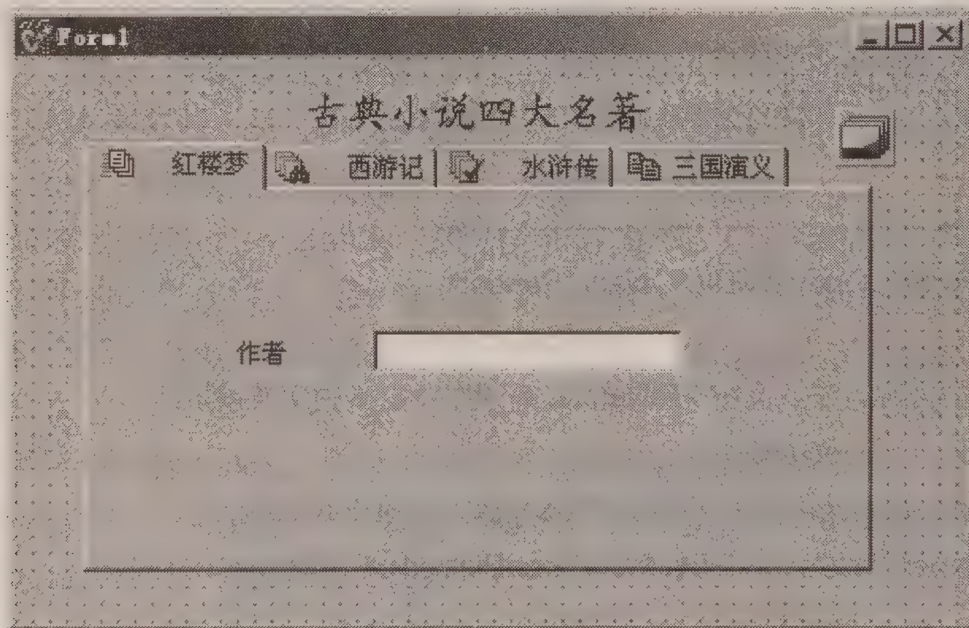


图 2-4　TTabControl 组件应用示例

2.3　实例创建分析

前文简要介绍了 C++ Builder 中的文本编辑组件和 MDI 多文档界面技术。现在，我们将应用这些知识开发一个实用应用程序—— BCB 书写器（BCBEditor）。该程序是一个基于 TRichEdit 组件，支持纯文本格式（*.txt）和多文本格式（*.rtf）的文本编辑器程序。

BCB 书写器程序以 Windows 95/98 的写字板（WordPad）为蓝本。首先我们将在 BCB 书写器逐一实现 WordPad 具备的必要功能，然后加入一些 WordPad 并不具备的高级功能，以弥补 Windows 95/98 写字板的不足。

作为一个完整而又有特色的文本编辑器程序，将逐步实现以下的部分：

- MDI 的编辑环境。
- 类似于 office 97 的停驻风格的工具条。
- 创建打开、编辑、保存文件。
- 复制、粘贴、剪切字符串。
- 设置文件字体属性。
- 打印文件。
- 替换文件中指定的字符串。
- 段落格式控制。
- 历史打开文件列。
- 显示当前所在行列。
- 主窗体的漂亮的背景贴图。

BCB 书写器如图 2-5 所示。

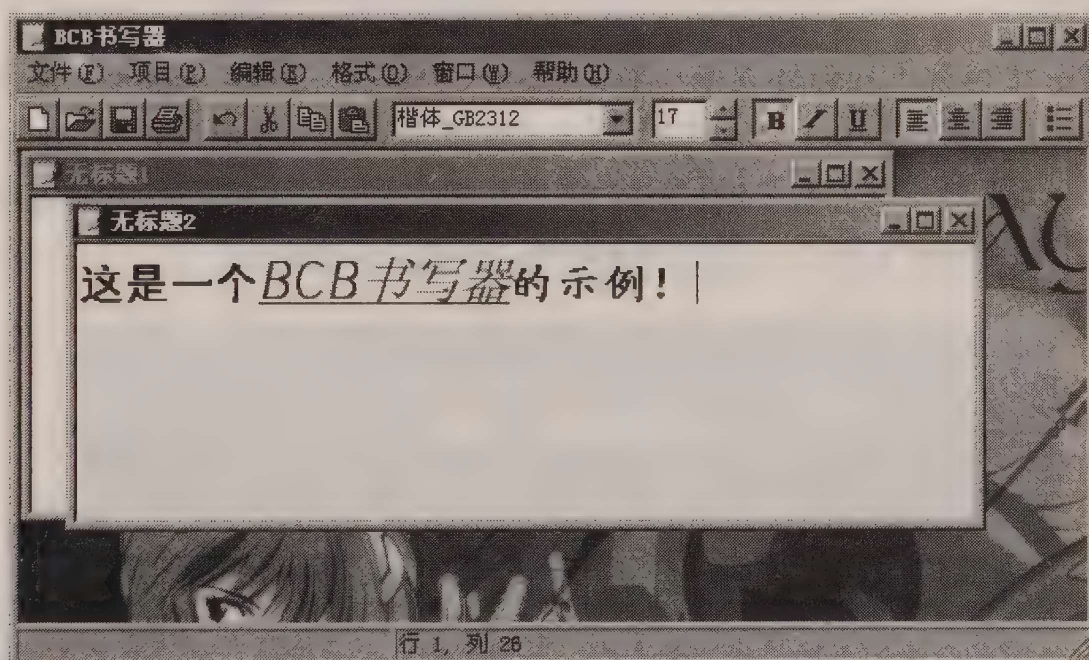


图 2-5 功能齐全、MDI 界面的 BCB 书写器

BCBEditor 工程中主要包含下面的一些文件：

- 主窗体单元（MDIForm）。类名：TMainForm，文件：Main.h，Main.cpp。
- 子窗体单元（MDIChild）。类名：TChildForm，文件 Child.h，Child.cpp。

子窗口的创建和总体控制功能在主窗体单元部分实现，同时主窗体单元完成一部分文本处理的功能，而另一些文本编辑和处理的功能在子窗体单元部分得以实现。此外，本程序还包含一个 About 窗体（类名：TAboutBox，文件：About.h，About.cpp）。

通过完成 BCB 书写器的范例程序，你将更进一步的了解和掌握文本编辑处理的各个方面的技巧，同时也将对 MDI 多文档应用程序有更深刻的认识。

2.4 创建 MDI 的编辑环境

在设计阶段，可创建 MDI 父窗体作为应用程序主窗体，同时可视化创建 MDI 程序的子窗

体。创建 MDI 应用程序的基本步骤是：

(1) 创建主窗口类。

(2) 创建子窗口类。

(3) 设计菜单并处理融合菜单。

(4) 运行时创建子窗口。

如前所述，在 C++ Builder 中创建主窗口类和子窗口类是很容易的。主窗口类和子窗口类都是一个普通的窗体，对这类两个窗体都可进行一般的可视化设计。唯一特别的是需要将主窗体的 FormStyle 设为 fsMDIForm，子窗体的 FormStyle 设为 fsMDIChild。而后，在程序中实例化子窗体就可以完成一个 MDI 多文档界面的创建。

2.4.1 主窗体和子窗体界面

1. 主窗体界面

主窗体界面主要包含了主窗体菜单，状态栏和一个快捷工具栏，以及一系列非可视化组件，包括 ImageList 组件，用于工具栏图标；OpenDialog 对话框组件，用于打开文件；SaveDialog 组件，用于存档文件；PrintDialog 组件，用于打印文档；PrinterSetupDialog 组件，用于打印机设置；FontDialog 组件，用于字体设置，如图 2-6 所示。

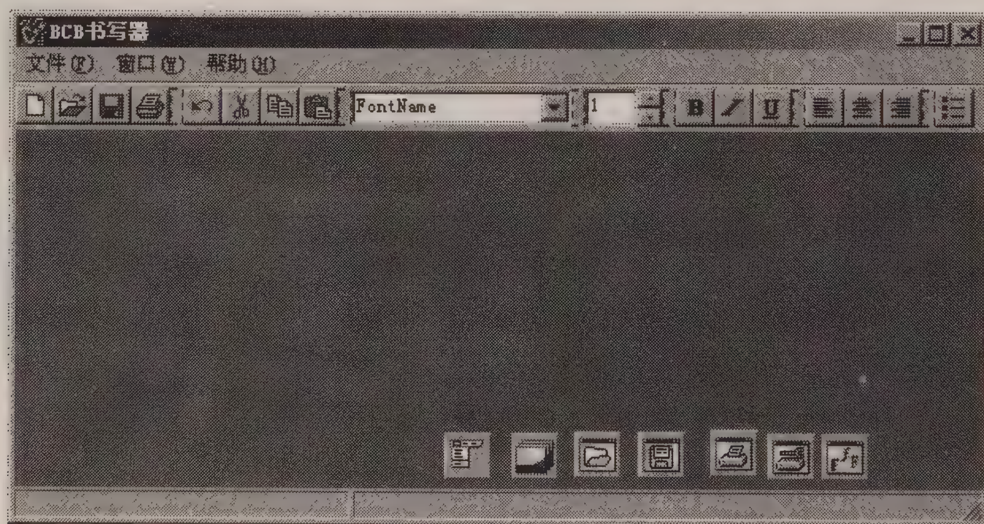


图 2-6 BCB 书写器范例程序的主窗体界面设计

工具栏上包含一个 ComboBox 组合框组件，用于设置字体名称；一个 Edit 组件和一个 UpDown 组件，这两个组件组合应用设置字体大小。Edit 组件和 UpDown 组件组合使用的方法是将 UpDown 组件的 Associate 属性设为 Edit 组件的名字，在本程序中：

```
UpDown1->Associate=FontSize;
```

同时，UpDown 组件还可以实现文本框组件的数字范围，在本例中：

```
UpDown1->Max=1638;
UpDown1->Min=1;
```

对于工具栏上用于控制加粗、倾斜、下划线以及段落和项目符号的七个按钮，应将 Style 属性设为 tbsCheck，以便这些按钮具备按下和未按下两种状态。在程序中可以通过按钮的 Down 属性获取或设定这些按钮的状态。

另外对于工具栏控制段落对齐的一组按钮，应该将这些按钮 Grouped 属性设为 true，同时将 AllowAllUp 属性置为 false。这两项设定的目的是使这组按钮能够具有 RadioButton 的性能，即同一时刻有且仅有一个按钮处于按下状态。

最后，作为 MDI 程序主窗体，窗体 MainForm 的 FormStyle 属性应设为 fsMDIForm。

2. 子窗体界面

子窗体是进行文本编辑的窗体，如图 2-7 所示。

子窗体部分担负着进行文本编辑、文本控制以及查找和替换等功能。其中文本编辑框是一个 RichEdit 组件，其属性 Align 应该设为 alClient，以便在子窗口大小改变时，文本编辑框能够始终占满窗口；ScrollBars 属性设为 ssVertical。子窗体还包括用于查找替换的对话框组件 FindDialog1 和 ReplaceDialog1。

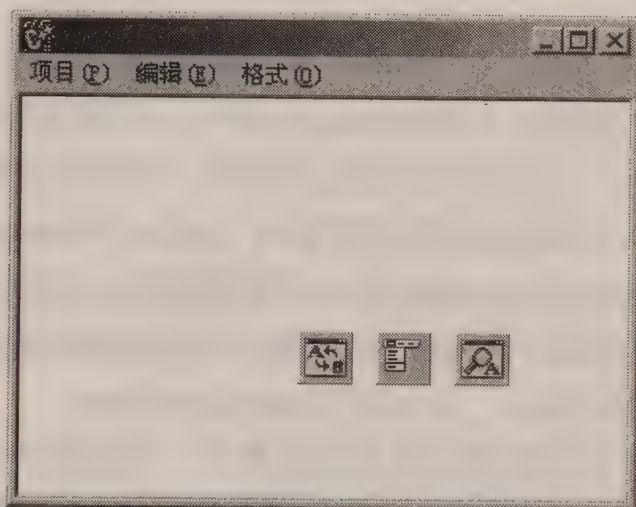


图 2-7　范例程序子窗体界面设计

如前所述，子窗体 ChildForm 的 FormStyle 属性应设为 fsMDIChild。

另外，对于 MDI 程序的子窗体，需要在运行期间动态创建子窗体类示例，所以应选择【**Project/Options**】菜单，在 Forms 页的自动创建列表对话框中设置子窗体特性为 Available Form。

2.4.2　实现可停驻（Docking）工具条

Borland C++ Builder 从它的第 4 版开始对停驻（Docking）技术提供编程支持。在 C++ Builder 中，理论上任何可视组件（继承于 TControl 类的组件）都可以实现停驻。在 BCB 书写器范例中，我们也将实现具有停驻风格的工具条。

可以想象，从底层实现 Docking 并不容易。但这种情况不会在 C++ Builder 中发生，因为 C++ Builder 隐藏了实现停驻的复杂工作。对于程序开发人员来说，仅需要做简单的设置就可以实现停驻功能。

停驻技术涉及两个对象——实施停驻操作的组件和接受停驻组件的容器控件。容器方面，C++ Builder 的停驻功能不仅仅支持了与窗体相连，而且支持与一般控件（继承于 TWinControl 的组件）相连，也就是说，可以停驻于任何控件容器上。面板（Tpanel）、窗体（TForm）、CoolBar 和 ControlBar 等都可以通过设置它们的 DockSite 属性成为停驻容器。同

时还可以设置这些容器的 AutoSize 属性为 true，这样可以使它们在实际管理停驻控件时才显示自己。

对于需要实现停驻的组件，编程中需要将其 DragKind 属性设置为 dkDock，同时将其 DragMode 属性设置为 dmAutomatic。这样这个组件就具备了从当前位置拖放并停驻到某个容器的能力。

同时，C++ Builder 5 提供了一系列与停驻相关的事件，用于跟踪用户停驻操作。对于实施停驻的组件有 OnStartDock、OnEndDock 事件，对于接受停驻的控件有 OnDragOver 和 OnDragDrop 事件。

C++ Builder 5 提供完整的属性和方法用于控制停驻行为。对于实施停驻的组件，可以使用 Dock、ManualDock 与 ManualFloat 方法移动停放组件的位置。对于容器来说，有一个 DockClientCount 属性用于获得停放组件的数目，以及一个 DockClients 属性用于访问停驻的组件。这个属性是这样声明的：

```
_property TControl* DockClients[int Index]={read=GetDockClients};
```

另一方面，可以使用停驻容器的 DockManager 属性。该属性实现 IDockManager 接口。通过该接口可实现高级停驻控制，它具有很多有用的特性，可用来定制停驻容器的功能，甚至可以支持流化容器状态。使用 DockManager 属性需要将 UseDockManager 属性设置为 true。

停驻组件还具有一个 FloatingDockSiteClass 的重要属性，它可以定制停驻组件在浮动时的窗体类型。一般可以设置为 TCustomDockForm，例如可以这样设置：

```
ComboBox1->FloatingDockSiteClass=__classid(TCustomDockForm);
```

该窗体类型是一个专为停驻功能预定义的窗体类型，其边框类型为 bsToolWindow。

在 BCB 范例中，创建可停驻工具条的工作是简单的。程序中使用 TControlBar 组件作为停驻容器，而工具条 TToolBar 组件作为实施停驻的组件，见图 2-8。下面是 DFM 文件的

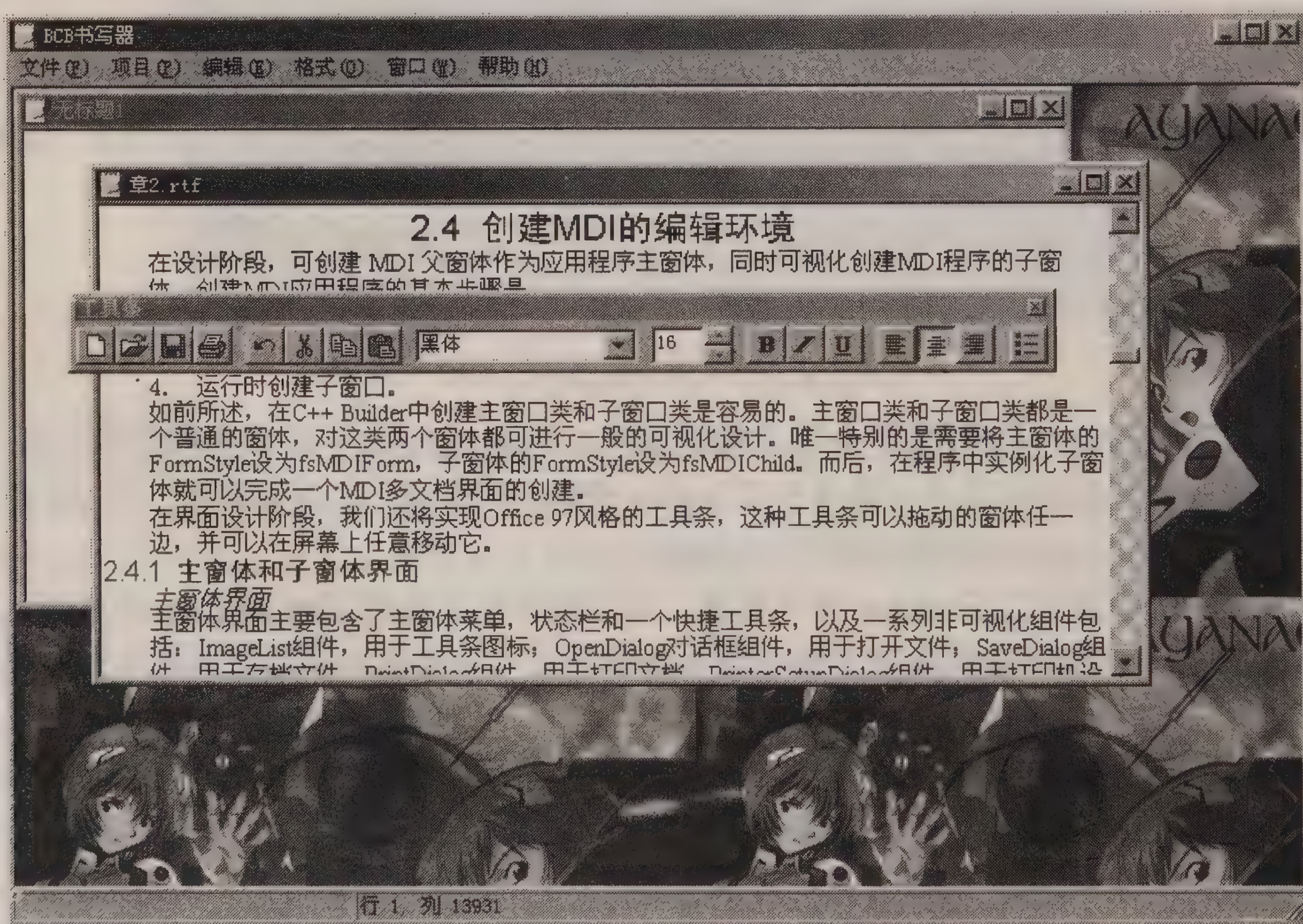


图 2-8　在 BCB 书写器中实现可停驻工具条

相关代码：

```
object ControlBar1:TControlBar
    Align=alTop
    AutoSize=True
    AutoDock=True
    DockSite=True
    object ToolBar1:TToolBar
        AutoSize=True
        DragKind=dkDock
        DragMode=dmAutomatic
    ……
```

2.4.3 菜单融合处理和窗体布局控制

1. MDI 菜单融合处理

父窗口的菜单应作为应用程序主菜单。如果子窗口有菜单， 则当子窗口在运行获得焦点时，子窗口的菜单项将融合父窗口菜单。例如本例中融合前的菜单如图 2-6、图 2-7 所示，而图 2-9 是融合后的菜单。

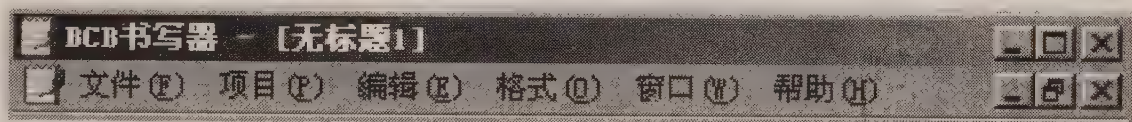


图 2-9　融合后的菜单，其中文件、窗口和帮助来自主窗体，其他三项来自子窗体

创建父窗口与子窗口菜单的方法与创建普通窗体菜单相同。菜单融合是指程序运行过程中，子菜单与父窗口菜单的相互作用。 如当子窗口获得焦点时，子窗口的菜单或插入主窗口的菜单中，或将替换部分或全部的父窗口菜单。菜单融合是 C++ Builder 自动实现的。

与菜单融合紧密相关有的两个属性：

- 窗体的 Menu 属性。
- 顶级菜单项的 GroupIndex 属性。

Memu 属性用于描述窗体的主菜单，也就是进行菜单融合的对象。顶级菜单的 GroupIndex 属性决定出现在菜单栏中各菜单项的位置。在菜单融合中，GroupIndex 将决定融合菜单是插入还是替换主窗体菜单栏中的菜单。

GroupIndex 的缺省值是 0，其值决定融合后菜单的布局，具体可由下面规则确定：

- 数值越小，菜单的位置越靠左。例如：GroupIndex 为 0 的菜单将出现在菜单栏中的最左端。随着 GroupIndex 数值的增大，菜单项依次向右排列。
- 若需替换主菜单中的某一菜单项，则应设置子菜单相应菜单项与须替换菜单项的 GroupIndex 为相同值。这条规则适合一个或多个菜单项。例如，主菜单中的“Edit”菜单项的 GroupIndex 的值为 1，将子菜单的一个或多个菜单项的 GroupIndext 的值设

为 1，则在运行时，这些菜单项替换主窗口的“Edit”菜单。若将同一窗体的多个菜单项的 GroupIndex 设为相同值，原有的排列顺序在菜单融合时将保持不变。

- 若要在菜单融合时插入菜单项，需在主菜单中预留数值“位置”。例如，主菜单的菜单项数值为 0，9，9，则子菜单 GroupIndex 数值为 1，2，2 的菜单在融合时将插入其中。

除了菜单融合之外，C++ Builder 生成 MDI 多文档应用程序还能够合并主窗口和子窗口的标题。在子窗体获得焦点并最大化时，主窗口和子窗口的标题将被合并。

2. 子窗口布局控制

在本章创建的“BCB 书写器”程序中，存在一个类似 Mircosoft Word 的标题名为“窗口”的顶级菜单项，这个菜单项包含子窗口布局控制选项和子窗口的快捷切换。

C++ Builder 的 TForm 类提供了几个对象方法用于对子窗口布局进行控制：

Cascade	子窗口重叠。
Tile	子窗口平铺。
ArrageIcons	重排子窗口最小化时的图标。
Minimize	所有子窗口最小化。

对于 Tile 方法，可以由 TForm 类的 TileMode 属性进行控制。它有两个值：tbHorizontal 和 tbVertical，分别表示横向平铺和纵向平铺。

在响应函数中分别调用这些方法，即可实现子窗口布局控制功能。

```
void __fastcall TMainForm::Tile1Click(TObject *Sender)
{     // 菜单项横向平铺的响应函数
      TileMode = tbHorizontal;
      Tile();
}
```

而子窗口的快捷切换的实现也是简单的。这项功能可以交由 C++ Builder 自动实现。方法是将 MainForm 的 WindowMenu 属性设为欲显示子窗口列表的菜单项。例如在本例中：

```
MainForm->WindowMenu=Window1
```

2.5 基本文本编辑功能的实现

在本节中将具体实现“BCB 书写器”所应具有的基本功能，包括文档的打开、存盘及编辑等。

2.5.1 文档的打开、存盘、关闭和打印

新建一个文档时，需要为该文档新建一个子窗口。假设子窗口类已经编写完毕，就可以通过创建子窗口类的实例新建子窗口。另外在打开文件时也需要新建一个子窗口。新建子窗口的同时，子窗口对象要求接受一个标题名字符串，该字符串可能代表一个实际文件，通过

FileExists 函数确定文件是否在硬盘上，若文件确实存在，则用该子窗体打开这个文件。程序如下：

```
void __fastcall TMainForm::CreateChild(AnsiString Name)
{
    TChildForm *Child;
    Child = new TChildForm(Application);
    Child->Caption = Name;
    if (FileExists (Name)){
        Child->isNamed = true;
        Child->OpenFile(Name);
    }
    else Child->isNamed = false;
}
```

其中 isNamed 是子窗体类的一个属性，该属性与文档存盘操作相关。如果 isNamed == false，则在用户进行存盘操作时，执行 SaveAs。

通过调用 CreateChild 函数完成新建和打开操作。当用户选取菜单上的新建选项或单击工具栏上的新建按钮，将新建文档。其响应函数为：

```
void __fastcall TMainForm::New1Click(TObject *Sender)
{
    CreateChild("无标题" + IntToStr(MDIChildCount + 1));
}
```

MDIChildCount 是 TFrom 类的一个属性，它用于返回当前子窗体的个数。

当用户选取菜单上的打开选项或单击工具栏上的打开按钮，将激活 OpenDialog1 对话框，并打开文档。其响应函数为：

```
void __fastcall TMainForm::Open1Click(TObject *Sender)
{
    if (OpenDialog->Execute())
        CreateChild(OpenDialog->FileName);
}
```

存盘功能并不涉及子窗体的创建，所以更适合在子窗体类中实现。在子窗体文件 Child.cpp 中，子窗体菜单的“存盘”和“另存为”选项的响应函数分别为：

```
void __fastcall TChildForm::Save1Click(TObject *Sender)
{
    if(isNamed){
        RichEdit1->Lines->SaveToFile(cFileName);
        RichEdit1->Modified = false;
    }
    else{
        Saveas1Click(Sender);
    }
```

```
}
void __fastcall TChildForm::Saveas1Click(TObject *Sender)
{
        MainForm->SaveDialog->FileName = cFileName;
        If (MainForm->SaveDialog->Execute() ){
                cFileName= MainForm->SaveDialog->FileName;
                Caption = ExtractFileName(cFileName);
                isNamed=true;
                Save1Click(Sender);
        }
}
```

cFileName 是子窗体类的属性，用于纪录当前编辑的文件名，也有可能为“无标题”。请注意 Save1Click、Saveas1Click 两个函数的逻辑关系，它们是互相调用的。

关闭文档操作需要关闭与之对应的子窗口。文档关闭时需要实现存盘提示和历史打开文件信息添加，这涉及到子窗体类的 OnClose 事件和 OnCloseQuery 事件的处理。填写子窗体的 OnClose 事件的响应函数如下：

```
void __fastcall TChildForm::FormClose(TObject *Sender,
      TCloseAction &Action)
{
        // 进行历史打开文件的处理，将在稍后讲述
        if (isNamed) {
                MainForm->HistoryList->Add(cFileName);
                MainForm->BuildHistory(MainForm->File1);
        }
        // 使得关闭子窗体命令时，释放内存并关闭
        Action=caFree;
}
```

Action 属性的值为 TCloseAction 枚举类型，其值为 caNone、caHide、caFree、caMinimize，表示窗体的不同行为，如隐藏、销毁、最小化等。

填写子窗体的 OnCloseQuery 事件的响应函数，此事件在关闭一个窗体时触发，用于决定是否确实进行窗口关闭操作。当用户发出关闭命令时，此函数判断用户是否修改了编辑框中的内容而没有存盘。如果是，弹出对话框提示用户是否进行存盘操作，如图 2-10 所示。

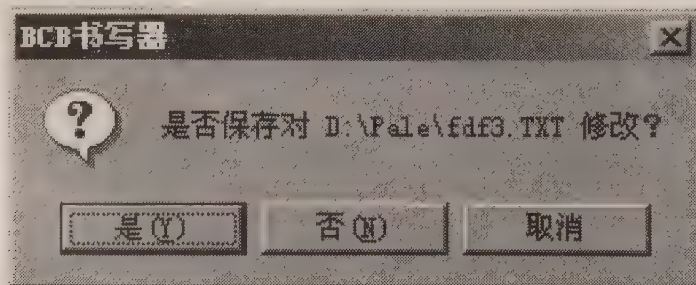


图 2-10　MessageBox 函数生成的对话框

```
void __fastcall TChildForm::FormCloseQuery(TObject *Sender, bool &CanClose)
{
    if (RichEdit1->Modified){
        switch (MessageBox(Handle,("是否保存对 "+cFileName+" 修改？").c_str(),
            MainForm->Caption.c_str(),MB_YESNOCANCEL|MB_ICONQUESTION))
        {
        case IDYES:
            Save1Click(NULL);
        case IDCANCEL:
            CanClose=false;
        }
    }
}
```

这里用到了 MessageBox 函数，这个函数显示出标准对话框，此函数定义如下：

```
int MessageBox（
  HWND hWnd,        // 父窗口句柄
  LPCTSTR lpText,   // 文本
  LPCTSTR lpCaption,    // 标题文本
  UINT uType            // MessageBox 风格
);
```

hWnd 表示父窗体的句柄，lpText 指向对话框的文本内容字符串，lpCaption 指向对话框的标题字符串；uType 参数定义对话框的样式，包括按钮、图标和其他非常繁多的控制。下面举出按钮和图标的一些常用常量。

MB_ABORTRETRYIGNORE	包含 Abort, Retry,Ignore 三个按钮。
MB_OK	仅有 OK 一个按钮。
MB_OKCANCEL	包含两个按钮：OK 和 Cancel。
MB_RETRYCANCEL	包含两个按钮：Retry 和 Cancel。
MB_YESNO	包含两个按钮：Yes 和 No。
MB_YESNOCANCEL	包含 Yes、No 和 Cancel 三个按钮。
MB_ICONWARNING	显示警告图标。
MB_ICONINFORMATION	显示带有小写字母 I 的图标。
MB_ICONQUESTION	显示一个带问号的图标。
MB_ICONSTOP	显示停止图标。

MessageBox 函数的返回值常量为：

IDABORT	Abort 按钮被选择。
IDCANCEL	Cancel 按钮被选择。
IDIGNORE	Ignore 按钮被选择。
IDNO	No 按钮被选择。
IDOK	OK 按钮被选择。
IDRETRY	Retry 按钮被选择。
IDYES	Yes 按钮被选择。

另外，FormCloseQuery 函数中修改 CanClose 属性为 false，表示取消关闭窗口操作，将不再关闭子窗口。

实现打印 RichEdit 组件内的文档是简单的，使用 TRichEdit 组件提供的 Print 对象方法。响应 PrintButton 的 OnClick 事件：

```
void __fastcall TMainForm::PrintButtonClick(TObject *Sender)
{
    CurrentForm=(TChildForm *)ActiveMDIChild;
    if ( PrintDialog->Execute() )
        CurrentForm->RichEdit1->Print(CurrentForm->cFileName);
}
```

TForm 的 ActiveMDIChild 属性是当前获得焦点的子窗体的指针，将其强制转换为自定义的 TChildForm 类，即可对子窗体进行操作。

2.5.2 剪贴板编辑功能

在本例中，利用剪贴板实现编辑功能是简单的。这是因为 RichEdit 组件提供了整套相关的方法，这里所要作的只是在合适的时候进行调用。全选和删除功能也可由 TRichEdit 组件的相应方法来实现。

Source 实现剪贴板功能的相关代码

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::CopyButtonClick(TObject *Sender)
{
    // 实现“复制”功能
    CurrentForm=(TChildForm *)ActiveMDIChild;
    if (CurrentForm!=NULL)
        CurrentForm->RichEdit1->CopyToClipboard();
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::CutButtonClick(TObject *Sender)
{
    // 实现“剪切”功能
    CurrentForm=(TChildForm *)ActiveMDIChild;
    if (CurrentForm!=NULL)
        CurrentForm->RichEdit1->CutToClipboard();
}
//---------------------------------------------------------------------------
```

```
void __fastcall TMainForm::PasteButtonClick(TObject *Sender)
{
        // 实现“粘贴”功能
        CurrentForm=(TChildForm *)ActiveMDIChild;
        if (CurrentForm!=NULL)
                CurrentForm->RichEdit1->PasteFromClipboard();
}
//---------------------------------------------------------------------------
void __fastcall TChildForm::Selectall1Click(TObject *Sender)
{
        // 全选功能响应函数
        RichEdit1->SelectAll();
}
//---------------------------------------------------------------------------
void __fastcall TChildForm::Delete1Click(TObject *Sender)
{
        // 删除功能响应函数
        RichEdit1->ClearSelection();
}
//---------------------------------------------------------------------------
```

另外，文本编辑时的 Undo 功能也可以轻松实现，这里提供两种方案，一种是直接利用 TRichEdit 组件的 Undo 方法；另一种是向 RichEdit 对象发送消息。

```
void __fastcall TMainForm::UndoButtonClick(TObject *Sender)
{
        CurrentForm=(TChildForm *)ActiveMDIChild;
        if ( CurrentForm->RichEdit1->HandleAllocated() )
                SendMessage(CurrentForm->RichEdit1->Handle, EM_UNDO, 0, 0); // 通过发送消息
        //CurrentForm->RichEdit1->Undo(); // 直接使用 Undo 方法
}
```

向 TRichEdit 组件（本质上是 Windows 的 RichEdit 控件）发送消息是一种行之有效的方法。在这里可能显得大材小用，但对于 Bolord C++ Builder 没有封装的功能，或是 VCL 事件没有包括的 Windows 消息，就不得不用到 SendMessage 函数向相应控件发送消息来实现。下文中实现的“报告当前所在行列”的功能就属这种情况。

2.6 字体格式控制、查找与替换

本节涉及 RichEdit 文本编辑字体和段落控制的相关处理，以及实现编辑器查找、替换功能。

2.6.1 字体和段落格式控制

1. 响应 SelectionChange 事件

在 RichEdit 编辑框中，每一个字符都有自己的字体属性和格式属性。当用户改变光标位置或选择了字符时，需要将当前的字体属性和格式属性在工具栏上报告出来。比如说，当光标移动到设为粗体的字符时，工具栏的粗体按钮应变为按下状态。响应 TRichEdit 的 OnSelectionChange 事件实现该功能，具体代码为：

Source 处理 SelectionChange 事件代码

```
//---------------------------------------------------------------------------
void __fastcall TChildForm::RichEdit1Change(TObject *Sender)
{
        MainForm->BButton->Down =
                RichEdit1->SelAttributes->Style.Contains(fsBold);
        MainForm->IButton->Down =
                RichEdit1->SelAttributes->Style.Contains(fsItalic);
        MainForm->UlButton->Down =
                RichEdit1->SelAttributes->Style.Contains(fsUnderline);
        MainForm->BulletsButton->Down =
                bool(RichEdit1->Paragraph->Numbering);
        MainForm->FontSize->Text =
                AnsiString(RichEdit1->SelAttributes->Size);
        MainForm->FontName->Text = RichEdit1->SelAttributes->Name;
        switch((int)RichEdit1->Paragraph->Alignment){
        case 0:
                MainForm->LeftAlign->Down   = true;
                break;
        case 1:
                MainForm->RightAlign->Down  = true;
                break;
        case 2:
                MainForm->CenterAlign->Down = true;
                break;
        }
        // 实现显示当前行数和列数的函数，在本章最后一节具体讲解
        MainForm->ShowPos();
}
```

```
//---------------------------------------------------------------------------
```

RichEdit 的 SelAttributes 属性包含一个 Style 的集合类的属性，由它可以确定当前文本字体是否为斜体、粗体或下划线。Paragraph 属性的 Numbering 属性用以表示是否有项目符号，Alignment 属性表示当前段落的对齐方式；SelAttributes 属性的 Name 和 Size 属性分别包含了当前文本的字体名称和字体大小。

2. 字体控制功能

在 BCBEditor 范例程序中，为实现字体选择提供了两种方式，简单的控制使用工具栏上的组合框（ComboBox）设置字体名称，上下组件和文本编辑框设置字体大小，相应的按键用以设定字体的风格。对于选择字体名称的下拉框控件，首要问题是如何获得全部的字体名称。

为了获得字体列表，可以在 MainForm 的 OnCreate 响应函数中加入：

```
FontName->Items = Screen->Fonts;
```

这行代码通过全局 TScreen 类实例 Screen 获得当前系统支持的所有字体，并填入名为 FontName 的组合框中。

在进行文本控制和格式控制处理之前，首先加入一个函数，用于获得当前活动的子窗体中编辑框 RichEdit 的 SelAttributes 属性。

```
TTextAttributes *__fastcall TMainForm::SelText()
{
    CurrentForm=(TChildForm *)ActiveMDIChild;
    return CurrentForm->RichEdit1->SelAttributes;
}
```

响应组合框的 OnChange 事件，使程序支持用户由下拉框设置字体。

```
void __fastcall TMainForm::FontNameChange(TObject *Sender)
{
    SelText()->Name = FontName->Items->Strings[FontName->ItemIndex];
}
```

响应文本框的 OnChange 事件，使程序支持用户设置字体大小。

```
void __fastcall TMainForm::FontSizeChange(TObject *Sender)
{
    int fontsize = StrToInt(FontSize->Text);
    if (fontsize < 1){
        ShowMessage("输入整数应在 1 和 1638 之间。");
        FontSize->Text = 1;
    }
    else if (fontsize > 1638){
        ShowMessage("输入整数应在 1 和 1638 之间。");
        FontSize->Text = 1638;
    }
    if (ActiveMDIChild!=NULL)
        SelText()->Size = StrToInt(FontSize->Text);
```

```
}
```

由于 TrueType 字体支持的大小在 1 到 1638 之间，所以作相应的容错处理。

处理工具栏按钮 BButton、IButton 和 UlButton 的 OnClick 事件。

```
void __fastcall TMainForm::BButtonClick(TObject *Sender)
{
    if ( BButton->Down )
        SelText()->Style =SelText()->Style << fsBold;
    else
        SelText()->Style =SelText()->Style >> fsBold;
}
void __fastcall TMainForm::IButtonClick(TObject *Sender)
{
    if ( IButton->Down )
        SelText()->Style = SelText()->Style << fsItalic;
    else
        SelText()->Style = SelText()->Style >> fsItalic;
}
void __fastcall TMainForm::UlButtonClick(TObject *Sender)
{
    if ( UlButton->Down )
        SelText()->Style = SelText()->Style << fsUnderline;
    else
        SelText()->Style = SelText()->Style >> fsUnderline;
}
```

3. TFontDialog 字体对话框组件

BCB 书写器程序中为用户提供的另一种字体设定方式由子窗体菜单的“字体”菜单项进行设置，这里使用了 FontDialog 标准对话框。

FontDialog 对话框组件的基本属性是 Font，为 TFont 类型，可用来返回用户选择的字体。Device 是 TFontDialog 的另一个重要属性，用来确定字体列表的设备来源，有三种取值：fdScreen、fdPrinter 和 fdBoth，分别表示使用屏幕设备字体、使用打印机设备字体、同时从两种设备获取字体。此外，TFontDialog 组件允许定制输入和字体对话框外观和功能：MaxFontSize 和 MinFontSize 属性用来设定 FontDialong 的最大可用尺寸和最小可用尺寸；Options 属性为集合类型属性，它包含众多元素值用于控制 FontDialog 的外观和功能，例如，fdShowHelp 值可以使对话框显示帮助，TFontDialog 组件如图 2-11 所示。

通过标题名为“字体”的菜单项设置字型的实现函数如下：

```
void __fastcall TChildForm::Font1Click(TObject *Sender)
{
    MainForm->FontDialog1->Font=RichEdit1->Font;
    if (MainForm->FontDialog1->Execute())
```

```
        RichEdit1->SelAttributes->
            Assign(MainForm->FontDialog1->Font);
}
```

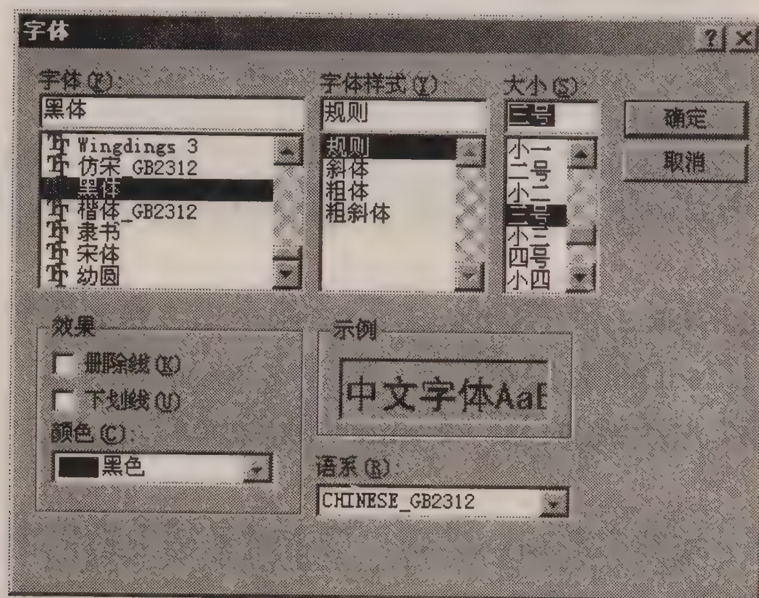


图 2-11　使用 TFontDialog 组件

4. 段落格式控制功能

另一方面，BCBEditor 程序提供对文本段落格式的支持。格式控制包括左对齐、中间对齐、右对齐和项目符号。在实现对齐功能的控制时，用到了利用响应函数的 Sender 参数实现响应函数重用的技术。具体来说，首先，在 MainForm 的 OnCreate 响应函数中对工具栏按钮组件设置标记：

```
LeftAlign->Tag=0;
RightAlign->Tag=1;
CenterAlign->Tag=2;
```

而 LeftAlign 按钮、RightAlign 按钮、CenterAlign 按钮用了同一个响应函数：

```
void __fastcall TMainForm::AlignBtnClick(TObject *Sender)
{
    CurrentForm=(TChildForm *)ActiveMDIChild;
    TControl *AliBtn = (TControl*)(Sender);
    CurrentForm->RichEdit1->Paragraph->Alignment=
        (TAlignment)AliBtn->Tag;
}
```

上述三个按钮 Tag 属性所设置的值与 Paragraph 的 Alignment 属性的值是对应的，利用将 Sender 参数强制转换获取按钮对象的指针，从而可以通过该指针对相应按钮对象实施控制，实现响应函数重用。

项目符号的实现与对齐相类似，具体代码为：

```
void __fastcall TMainForm::BulletsButtonClick(TObject *Sender)
{
    CurrentForm=(TChildForm *)ActiveMDIChild;
```

```
        CurrentForm->RichEdit1->Paragraph->Numbering =
            (TNumberingStyle)BulletsButton->Down;
}
```

2.6.2 查找与替换

查找和替换是各种编辑器必不可少的功能，利用强大的 C++ Builder，实现这项功能并不十分复杂。

C++ Builder 提供 TFindDialog 和 TReplaceDialog 两个用于查找和替换的标准对话框组件，并且 TReplaceDialog 是从 TFindDialog 继承的。

TFindDialog 的重要属性包括 FindText、Position 和 Option。

- FindText 属性。

此属性描述用户想要寻找的字符串。当用户单击对话框中的“查找下一个”按钮时，对话框内编辑框的内容被赋给 FindText 属性。

- Position 属性。

对话框的左上顶点，用于确定对话框弹出的位置。

- Options 属性。

此属性为集合类型，具体表示查找对话框（FindDialog）的外观和行为。就外观来说，frDisableMatchCase 、 frDisableUpDown 、 frDisableWholeWord 、 frHideMatchCase 、 frHideWholeWord、frHideUpDown 值分别表示把对话框中的两个 CheckBox（全字匹配和区分大小写）和一组 RadioButton（向上和向下）关掉（使之变灰）或隐藏。行为是指通过 Option 属性可以得到用户选定的选项。比如用户是按了查找按钮还是取消按钮，是否选择了全字匹配或区分大小写等，如图 2-12 所示。

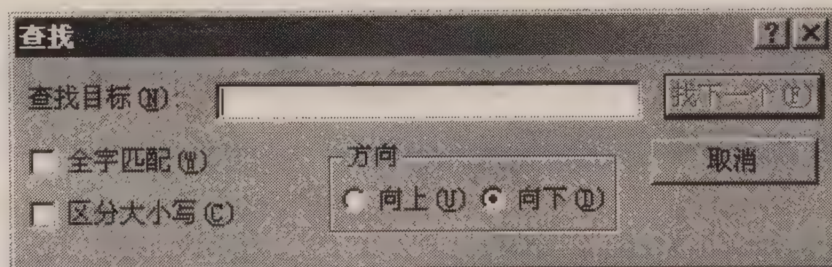


图 2-12 查找对话框

TReplaceDialog 继承自 TFindDialog，增加的属性只有 ReplaceText。此外 TReplaceDialog 的 Option 属性支持与替换操作相关的更多值。

需要说明的是，ReplaceDialog 和 FindDialog 的使用方式和其他标准对话框（如打开对话框等）有本质的不同。使用其他标准对话框组件的“固定套路”是：

```
if ( XXXXDialog1->Execute()
    …… （处理部分）
}
```

而使用 TReplaceDialog 和 TFindDialog 组件时，实际查找或替换处理代码需要在 OnReplace 或 OnFind 事件响应函数中执行。这是因为查找或是替换功能是在对话框还未关闭

时完成的，替换对话框如图 2-13 所示。

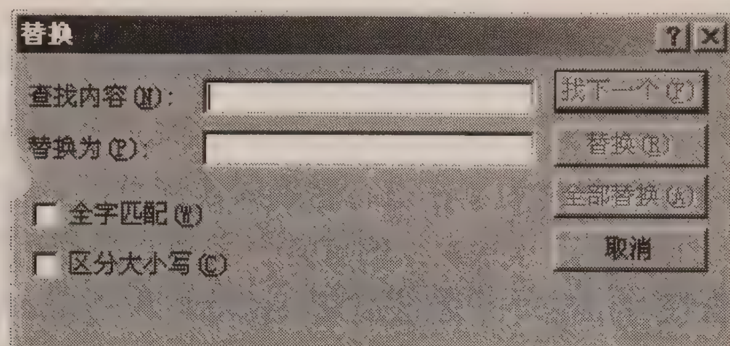


图 2-13　替换对话框

在程序中，菜单的查找选项激活查找对话框。

```
void __fastcall TChildForm::FindClick(TObject *Sender)
{
    FindDialog1->Position=Point(RichEdit1->Left + RichEdit1->Width/2,
    RichEdit1->Top+RichEdit1->Height/2);
    FindDialog1->Execute();
}
```

而真正的查找代码在 FindDialog1 的 OnFind 事件响应函数中。

Source 处理 OnFind 事件进行查找工作

```
//---------------------------------------------------------------------------
void __fastcall TChildForm::FindDialog1Find(TObject *Sender)
{
    TFindDialog *dlg=(TFindDialog *)Sender;
    int FoundAt, StartPos, ToEnd;
    if (RichEdit1->SelLength !  =0)
        StartPos = RichEdit1->SelStart+ RichEdit1->SelLength;
    else
        StartPos = RichEdit1->SelStart;
    ToEnd = RichEdit1->Text.Length() - StartPos;
    FoundAt = RichEdit1->FindText(dlg->FindText,
        StartPos, ToEnd, TSearchTypes()<< stMatchCase);
    if ( FoundAt != -1)
    {
        RichEdit1->SetFocus();
        RichEdit1->SelStart = FoundAt;
        RichEdit1->SelLength = dlg->FindText.Length();
    }
    else MessageBox(Handle,"没有找到符合条件的字符串!",
```

```
            "查找",MB_OK|MB_ICONINFORMATION);
        dlg->CloseDialog();
    }
//---------------------------------------------------------------------------
```

首先对 Sender 参数的强制转换是为了在进行替换时实现重用。查找时用到了 TRichEdit 的 SelStart 和 SelLength 属性，用于确定开始查找的位置。具体查找使用 RichEdit 对象的 FindText 方法。如果找到，将关闭查找对话框，并通过重置 SelStart 和 SelLength 属性将找到的内容选中。

“查找下一个”功能通过重用 FindDialong1Find 函数实现，可以这样处理：

```
void __fastcall TChildForm::FindNextClick(TObject *Sender)
{
    FindDialog1Find(FindDialog1);
}
```

在替换功能部分实现替换和全部替换，并且分别采用了两种不同的方式。

菜单的替换选项激活查找对话框。

```
void __fastcall TChildForm::ReplaceClick(TObject *Sender)
{
    ReplaceDialog1->Position =
        Point(RichEdit1->Left + RichEdit1->Width/2,
    RichEdit1->Top+RichEdit1->Height/2);
    ReplaceDialog1->Execute();
}
```

在 OnReplace 事件的响应函数中，包含具体进行替换操作的代码。

Source 处理 OnReplace 事件进行文本替换工作

```
//---------------------------------------------------------------------------
void __fastcall TChildForm::ReplaceDialog1Replace(TObject *Sender)
{
    if (ReplaceDialog1->Options.Contains(frReplace)){
        int SelPos = RichEdit1->Lines->Text.Pos(ReplaceDialog1->FindText);
        if (SelPos > 0)
        {
            RichEdit1->SelStart = SelPos - 1;
            RichEdit1->SelLength = ReplaceDialog1->FindText.Length();
            // Replace selected text with ReplaceText
            RichEdit1->SelText = ReplaceDialog1->ReplaceText;
        }
        else MessageBox(Handle,"替换工作已完成!",
```

```
                    "替换",MB_OK|MB_ICONINFORMATION);
        }
        else if (ReplaceDialog1->Options.Contains(frReplaceAll)){
                    int FoundAt, StartPos, ToEnd;
                    do{
                        if (RichEdit1->SelLength!=0)
                            StartPos = RichEdit1->SelStart+ RichEdit1->SelLength;
                        else
                            StartPos = RichEdit1->SelStart;
                        ToEnd = RichEdit1->Text.Length() - StartPos;
                        FoundAt = RichEdit1->FindText(ReplaceDialog1->FindText,
                            StartPos, ToEnd, TSearchTypes()<< stMatchCase);
                        if (FoundAt!=-1){
                            RichEdit1->SelStart = FoundAt;
                            RichEdit1->SelLength = ReplaceDialog1->FindText.Length();
                        RichEdit1->SelText = ReplaceDialog1->ReplaceText;
                        }
                    } while (FoundAt!=-1);
                    MessageBox(Handle,"替换工作已完成!","替换",
                        MB_OK|MB_ICONINFORMATION);
            }
        }
    }
//---------------------------------------------------------------------------
```

用户是选择了单一替换还是选择了全部替换由 Options 属性来判断。在实现单一替换中，演示了 Pos 方法的使用，这是一种简单的方式，但在全部替换中是不适合的，因为此方法每次都要查找全文。替换文本使用了 TRichEdit 的 SelText 属性。

从图 2-13 中可以看到，替换对话框中还包含一个查找的按钮，仍然可以重用查找对话框的处理函数，具体代码为：

```
ReplaceDialog1->OnFind = FindDialog1Find;
```

2.7　实现高级功能

通过上述工作，我们已经完成有关“BCB 书写器”全部必要的功能，至此，这个编辑器程序已经基本完整了，它已可以应付通常文本编辑工作的需要。但我们并不满足于此，在此将对 BCB 书写器程序作进一步的扩展，实现一些较为高级的功能。

本节的目标在于三点：实现历史文件打开菜单；实现当前行数、列数的报告；对父窗体的背景贴图。

2.7.1 历史文件列表菜单

历史文件列表菜单是应用程序中经常见到的一种功能，它的作用是使用户能够快捷地打开近来打开过的文件，“BCB 书写器”的历史文件列表菜单如图 2-14 所示。

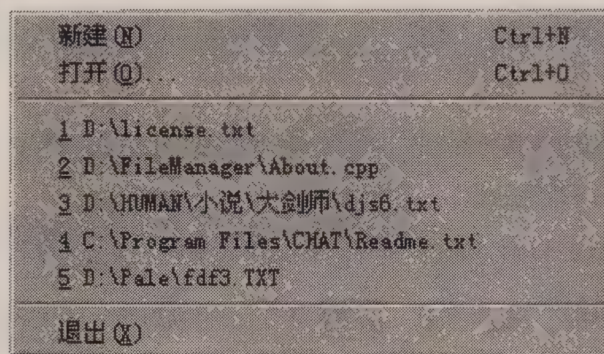


图 2-14　历史文件列表菜单

实现历史文件列表菜单的主要任务是存储用户新近打开的文件名，同时要用到动态调整菜单项的技术。

完成这项任务大致分成下面几个步骤：

(1)　一些必要的准备工作。包括常量、变量声明，及菜单插入标记的设定。

(2)　程序开始执行，读取存有历史文件列表的文件，并将历史文件存入一个 TStringList 类型变量 HistoryList。

(3)　重置菜单项，将历史文件构造成菜单。

(4)　完成用户单击历史文件菜单的响应函数。

(5)　当一个子窗体被关闭时，判断此子窗体是否有名（通过 isNamed），若有，则将名字插入 HistoryList 变量，并重新构造菜单。

(6)　当主窗体被关闭时，将当前的 HistoryList 变量存入文件。

下面具体来看看这项功能的实现过程。首先是准备工作，在主窗体文件声明常量：

```
const AnsiString HistoryListName="history";
const int MaxHistoryFile=5;
```

声明变量：

```
TStringList *HistoryList;
Int ShowHistoryFiles;
```

最后应对主窗体菜单的“文件”菜单项中的“关闭”子菜单项上方的分割行进行特殊命名，作为插入历史文件列表菜单的标记。在本例中，命名为 TopS。

其次，程序开始执行时，应读入存有历史文件列表的文件。

在 MainForm 的 OnCreate 事件函数中加入：

```
HistoryList=new TStringList();
AnsiString hFileName;
hFileName=ExtractFilePath(Application->ExeName) +HistoryListName;
if (FileExists(hFileName)){
```

```
        HistoryList->LoadFromFile(hFileName);
        BuildHistory(File1); // File1 是欲加入历史文件列表的菜单项名
}
```

然后，编写 BuildHistory 函数代码，实现菜单项的重新构造。

Source 根据历史打开文件动态构造菜单

```
//---------------------------------------------------------------------------
void TMainForm::BuildHistory(TMenuItem *FileMenu)
{
        TMenuItem *MenuItem;
        int Pos;
        // 将超出最大数目限制的历史文件去除
        if (HistoryList->Count>MaxHistoryFile)
        for (Pos = MaxHistoryFile;Pos<HistoryList->Count;Pos++)
                HistoryList->Delete(0);
        // BottomS!=NULL 表示菜单中已有历史文件列表，应进行删除。
        if (BottomS!=NULL){
                for (Pos=0;Pos<ShowHistoryFiles;Pos++)
                        File1->Delete(HistoryAddPos);
        }
        // 将新的历史文件列表加入菜单项
        if (HistoryList->Count>0){
                for (Pos=0;Pos<HistoryList->Count;Pos++){
                        MenuItem = new TMenuItem(FileMenu);
                        MenuItem->Caption="&"+IntToStr(Pos+1)+" "
                                +HistoryList->Strings[Pos];
                        MenuItem->OnClick = HistoryItemClick;// 响应函数
                        FileMenu->Insert(HistoryAddPos+Pos,MenuItem);
                }
                ShowHistoryFiles=HistoryList->Count;
                // 加入一条分割线
                if (BottomS==NULL){
                        BottomS =  new TMenuItem(FileMenu);
                        BottomS->Caption="-";
                        FileMenu->Insert(HistoryAddPos+HistoryList->Count,BottomS);
                }
        }
}
//---------------------------------------------------------------------------
```

```
void __fastcall TMainForm::HistoryItemClick(TObject *Sender)
{
        // 单击历史文件菜单的响应函数
        int Pos;
        AnsiString ItemName;
        TMenuItem *ParentItem;
        ParentItem=(TMenuItem *)((TMenuItem *)Sender)->Parent;
        Pos=ParentItem->IndexOf((TMenuItem *)Sender)-
         ParentItem->IndexOf(TopS)-1;
        ItemName=HistoryList->Strings[Pos];
        CreateChild(ItemName);
}
//---------------------------------------------------------------------------
```

HistoryItemClick 函数是所有历史打开文件菜单项的通用 OnClick 函数，该函数使用了 TMenuItem 类的 Parent 属性和 IndexOf 方法，用于获取用户敲击项在菜单中的位置。并根据位置打开相应的文件。

再后，对子窗口关闭和主窗口关闭事件进行相应处理，管理和存取历史打开文件的列表文件。

```
void __fastcall TChildForm::FormClose(TObject *Sender,
     TCloseAction &Action)
{
        if (isNamed){
                MainForm->HistoryList->Add(cFileName);
                MainForm->BuildHistory(MainForm->File1);
        }
        Action=caFree;
}
void __fastcall TMainForm::FormClose(TObject *Sender, TCloseAction &Action)
{
        HistoryList->SaveToFile(ExtractFilePath(Application->ExeName)+HistoryListName);
        delete HistoryList;
}
```

2.7.2 当前光标所在行、列数的报告

在 C++ Builder 中的 TRichEdit 类并未封装有报告当前所在行列的功能。实现这一功能需要用到向编辑框控件发送消息的技术。

以下是程序中 ShowPos 函数的代码。ShowPos 函数在 RichEdit 组件的 OnSelectionChange 事件中执行，以随时报告当前位置。

Source 实现当前光标行、列数的报告

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ShowPos()
{
        CurrentForm=(TChildForm *)ActiveMDIChild;
        if (CurrentForm!=NULL){
                int line =SendMessage(CurrentForm->RichEdit1->Handle,
                        EM_EXLINEFROMCHAR,CurrentForm->RichEdit1->SelStart,0);
                int lineindex=SendMessage(CurrentForm->RichEdit1->Handle,
                        EM_LINEINDEX,line,0);
                StatusBar->Panels->Items[1]->Text="行 "+IntToStr(line+1)+", 列 "
                        +IntToStr(CurrentForm->RichEdit1->SelStart-lineindex+1);
        }
}
//---------------------------------------------------------------------------
```

这里向 RichEdit 送出了两条消息：EM_EXLINEFROMCHAR 和 EM_LINEINDEX，作用是分别获取当前位置之前的行数，当前行之前的所有字符总数。通过简单计算即可获得目前所在的行数和列数，并在状态栏中显示出来。

SendMessage 函数是一个非常重要的函数。此函数涉及 Windows 消息机制的内容，关于 Windows 消息处理及编程问题在本书后面章节讲述，在此只对该函数作简单介绍。这个函数的定义为：

```
LRESULT SendMessage(
    HWND hWnd,        // 目标窗口句柄
    UINT Msg,             // 消息
    WPARAM wParam,        // 消息的 wParam
    LPARAM lParam         // 消息的 lParam
);
```

hWnd 参数代表发送对象的句柄。Msg 参数表明送出的消息类型，包括原有消息和自定义消息。WParam、lParam 参数是消息附加的信息。

函数的返回值因消息的不同而不同。

2.7.3 实现 MDI 父窗体的背景贴图

在 MDI 程序中，由于 MDI 父窗口一般的功能是提供子窗口显示的位置和提供菜单、工具栏、状态栏等，而窗口的客户区则一般不会有其他的用途，因此父窗口又被称为框架窗口。在父窗体的客户区绘制背景是一个不错的想法，但由于 MDI 程序的特殊性，这项工作并不像想象中那样简单。

直接在主窗口的 Canvas 上作画，画了半天，什么也显示不出来，在窗体上放个 TImage 组件也不行。实现这个目的只有通过重载窗口函数（Window Procedure）。

首先在 Main.h 头文件中声明变量：

```
Pointer OriginalClientProc;
Pointer ClientObjectInstance;
Graphics::TBitmap *BGBitmap;
```

然后，在 MainForm 的 OnCreate 事件响应函数中加入如下代码进行窗口函数重载：

```
ClientObjectInstance = MakeObjectInstance(ClientProc);
OriginalClientProc = (Pointer) SetWindowLong(ClientHandle,
      GWL_WNDPROC,(long)ClientObjectInstance);
BGBitmap=new Graphics::TBitmap();
BGBitmap->LoadFromFile(ExtractFilePath(Application->ExeName)+"bg.bmp");
```

我们知道，每一个 Windows 程序都必须具备窗口函数（Window Procedure），在 C++ Builder 中窗口函数由 C++ Builder 自动实现并且被隐藏起来。这里第一行代码是获取客户区窗体的原有窗口函数，第二行代码的目的是重载 MDI 父窗体的客户区窗体的窗口函数。这里用到了 SetWindowLong 函数，通过 SetWindowLong 函数可以改变窗口的风格（Style）、窗口函数地址、HINSTANCE 句柄和窗体的 ID 号。在我们的程序中，通过此函数实现窗口函数的重载。

重载的窗口函数编写如下：

```
void __fastcall TMainForm::ClientProc(TMessage & Msg)
{
      switch (Msg.Msg)
      {
            case WM_ERASEBKGND:
                  DrawClientWindow((HDC)Msg.WParam);
                  Msg.Result = true;
                  return;
            case WM_HSCROLL:
            case WM_VSCROLL:
                  Msg.Result = CallWindowProc((FARPROC)OriginalClientProc,
                        ClientHandle,Msg.Msg,Msg.WParam,Msg.LParam);
                  InvalidateRect(ClientHandle,0,true);
                  break;
            default:
                  Msg.Result=CallWindowProc((FARPROC)OriginalClientProc,
                        ClientHandle,Msg.Msg,Msg.WParam,Msg.LParam);
                  break;
      }
}
```

这里在处理 WM_ERASEBKGND 事件时用到了 DrawClientWindow((HDC) Msg.WParam)函数，在这个函数中进行实际的窗体背景绘制，绘制背景使用 Windows API 函数 BitBlt 平铺绘制

了 BGBitmap 图像。实现这个函数的代码如下：

```
void __fastcall TMainForm::DrawClientWindow(HDC & Hdc)
{
    TRect rect;
    ::GetClientRect(ClientHandle,(RECT *)&rect);
    intlefttop;
    // 使用 Windows API 函数 BitBlt
    for (int i=0; j<ClientHeight/BGBitmap->Height+1;i++)
    for (int j=0;j<ClientWidth/BGBitmap->Width+1;j++){
        left=j*BGBitmap->Width;
        top=i*BGBitmap->Height;
        ::BitBlt(Hdc,left,top,left+BGBitmap->Width,top+BGBitmap->Height,
        BGBitmap->Canvas->Handle,0,0,SRCCOPY);
}
```

最后，不要忘记在程序结束时作善后处理工作。在 OnClose 响应函数中添加：

```
SetWindowLong(ClientHandle,GWL_WNDPROC,(long)OriginalClientProc);
FreeObjectInstance(ClientObjectInstance);
delete BGBitmap;
```

至此已完成了 MDI 父窗体背景的绘制。但是，因为背景图案很大，频繁的重画会造成闪烁，为了消除闪烁，需要用到消息映射的技术。有关消息映射内容，在本书后面章节还将作具体的讨论。

在头文件 TMainForm 类的声明中加入：

```
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_ERASEBKGND,TWMEraseBkgnd,WMEraseBkgnd)
END_MESSAGE_MAP(TForm)
```

同时，完成 WMEraseBkgnd(TWMEraseBkgnd & Msg)函数，用于处理消息。

```
void __fastcall TMainForm::WMEraseBkgnd(TWMEraseBkgnd & Msg)
{
    Msg.Result = false;
}
```

这样做是告诉 Windows 重画时不必擦除背景，从而达到消除闪烁的目的。

至此，BCB 书写器编写完毕，从中读者可以领略到 C++ Builder 的强大的功能。

第3章

完整的文件管理器
——文件操作和文件流

C++ Builder 通用文件操作
与文件相关的可视化组件
Win32 风格文件管理器的实现
内存流和文件流
文件锁定和 Shell API

本章导读

每个程序员必须完成的常见任务之一就是操纵文件。Win32 操作系统引入长文件名文件系统以替代 DOS 时代的 8.3 格式。同时 Win32 提供整套支持 32 位文件操作的函数，Win32 API 文件函数能使应用程序访问文件而不管底层文件系统机制。C++ Builder 也提供一套新的支持 Win32 文件操作的函数，并且更加易于使用。

本章通过一个出色范例程序——BCB 文件管理器程序，讲述 C++ Builder 中文件操作编程的种种技巧。BCB 文件管理器和 Windows 95 的资源管理器几乎完全相同，提供 Win32 风格的文件信息检视和文件操作功能（包括拷贝、删除、剪切等等）。另外，本章扩展部分还将讲述 C++ Builder 文件流，与文件相关的重要 API 函数以及 Win32 外壳 API 等重要编程技术。

本章内容包括：

- 通用的文件操作，文件输入和输出。
- C++ Builder 中用于文件操作的可视化组件。
- 实现范例程序。并通过整个实例的实现，讲述进行文件查找、文件管理的方法，以及完成此文件管理器所必须的关键组件 TTreeView 和 TListView 组件。
- C++ Builder 的数据流化技术，文件流和内存流。
- 深入的部分：文件锁定与解锁 Shell API 方面的内容。并将利用 Shell API 实现 BCB 文件管理器的第 2 版本。

3.1 C++ Builder 的文件操作支持

Borland C++ Builder 是一个 C++的扩展集，在 C++ Builder 中依然可以使用 C/C++中标准输入输出（stdio.h）函数来进行文件操作，并继续沿用原来的方法，包括 fread、fwrite、fprintf 等等。这没有任何问题，但是在 Windows 编程中，应用为 Win32 文件系统设计的新函数是一个更加不错的选择。

Borland C++ Builder 提供一整套文件操作函数（来自 VCL），这些文件和目录相关函数的定义包含在 SysUtils.hpp 头文件中。具体来说，C++ Builder 中关于文件（包括目录）操作的主要函数有：

FileOpen	打开一个文件。
FileCreate	建立一个文件。
FileRead	从文件读取数据。
FileWrite	向文件写数据。
FileSeek	文件定位。
FileClose	结束文件操作，关闭一个文件。
FileAge	返回文件的修改日期和时间。
FileExists	检查特定文件是否存在。
FileGetDate	获得文件的 DOS 时间日期。
FileSetDate	设置文件的 DOS 时间日期。
FileGetAttr	获得文件的属性（如只读、隐藏等）。
FileSetAttr	设置文件的属性。
FindFirst	用于文件或目录查找。
FindNext	用于文件或目录查找。
FindClose	用于文件或目录查找。
DeleteFile	从磁盘删除一个文件。
RenameFile	重命名一个文件。
ChangeFileExt	改变文件的扩展名。
ExtractFilePath	返回文件的全路径。
ExtractFileDir	与 ExtractFilePath 基本相同，其结果少最后的"\"。
ExtractFileDrive	返回文件所在驱动器。
ExtractFileName	返回文件名（去除路径）。
ExtractFileExt	返回文件扩展名。
ExpandFileName	给定一个存在的文件名，返回它的包含路径的全名。
FileSearch	寻找一个特定文件的 DOS 路径。
DiskFree	返回指定驱动器的剩余容量。
DiskSize	返回指定驱动器的总容量。
FileDateToDateTime	将 DOS 时间格式的时间转换为 C++ Builder 时间格式。
DateTimeToFileDate	将 C++ Builder 时间格式的时间转换为 DOS 时间格式。
GetCurrentDir	获得当前路径。
SetCurrentDir	设置当前路径。
CreateDir	新建一个目录。
RemoveDir	删除一个目录。

C++ Builder 提供的文件操作函数相当丰富，其中 ExtractFileDir、ExtractFileDrive、ExtractFileName、ExtractFileExt 等确定文件属性的函数以及 FindFirst、FindNext、FindClose 文件查找操作的函数在标准的 C 函数库中并没有类似功能的函数与之对应。这些函数为程序员带来了极大的方便，在本章范例程序——BCB 文件管理器中包含这些函数的具体应用。

3.1.1 建立、打开和关闭文件

在 C++ Builder 中可以使用如下方法建立一个文件：

```
if ((iFileHandle =FileCreate("D:\\MyProject\\MyFile.sss"))==-1)
     MessageBox(0,"Error!","FileCreate",MB_OK);
```

对于一个建立成功的文件，就可以使用 FileRead、FileWrite 或 FileSeek 等函数对文件进行操作了。操作完成的文件使用 FileClose 函数关闭。

```
FileClose (iFileHandle);
```

对于已经存在的文件进行操作使用 FileOpen 函数，例如：

```
iFileHandle = FileOpen(OpenDialog1->FileName, fmOpenRead);
```

这个函数声明如下：

```
int __fastcall FileOpen(const AnsiString FileName, int Mode);
```

其中 Mode 参数可以有以下的取值：

fmOpenRead	只对文件进行读操作。
fmOpenWrite	只对文件进行写操作。
fmOpenReadWrite	对文件进行读写操作。
fmShareExclusive	对文件共享读写禁止。
fmShareDenyWrite	对文件的共享读操作被禁止。
fmShareDenyRead	对文件的共享写操作被禁止。
fmShareDenyNone	允许共享所有类型的操作。

在 Win32 中要建立新的文件或打开已有的文件，还可以使用 Win32 API 函数 CreateFile。CreateFile 提供了更多的选项，适合较高层次的需要，CreateFile 的原型为：

```
HANDLE CreateFile(
   LPCTSTR lpFileName,
   DWORD dwDesiredAccess,
   DWORD dwShareMode,
   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
   DWORD dwCreationDistribution,
   DWORD dwFlagsAndAttributes,
   HANDLE hTemplateFile
)
```

CreateFile 函数的详细说明如下：

lpFileName　　为指向文件名的指针。

dwDesiredAccess　打开放式，其值为 GENERIC_WRITE 或 GENERIC_READ。

dwShareMode　共享方式，为 FILE_SHARE_READ 或 FILE_SHARE_WRITE。

lpSecurityAttributes　指向安全属性指针。

dwCreationDistribution　创建方式，可选值为：

CREATE_NEW	建立新文件，若文件已存在则操作失败。
CREATE_ALWAYS	建立新文件，若文件已存在则覆盖已有文件。
OPEN_EXISTING	打开已存在文件。
OPEN_ALWAYS	打开文件，若文件不存在则建立一个新文件。
TRUNCATE_EXISTING	打开已有文件，并将文件内容清空。

dwFlagsAndAttributes　文件属性，可选值为：

FILE_ATTRIBUTE_ARCHIVE	归档文件。
FILE_ATTRIBUTE_COMPRESSED	压缩文件。
FILE_ATTRIBUTE_HIDDEN	隐藏文件。
FILE_ATTRIBUTE_NORMAL	无特殊属性文件。
FILE_ATTRIBUTE_READONLY	只读文件。
FILE_ATTRIBUTE_SYSTEM	系统文件。
FILE_ATTRIBUTE_TEMPORARY	临时文件。

hTemplateFile　指向临时文件的指针，临时文件提供打开或创建文件的属性和扩展属性。

下面的一行代码是利用 CreateFile API 函数打开文件的例子：

```
iFileHandle = CreateFile("D:\\MyProject\\MyFile.sss",
    GENERIC_READ, 0 ,NULL,OPEN_EXISTING,
    FILE_ATTRIBUTE_ARCHIVE| FILE_ATTRIBUTE_READONLY,NULL);
```

3.1.2　文件的读写操作

C++ Builder 提供了 FileRead 和 FileWrite 函数进行文件的读写操作。FileRead 函数定义为：

```
int __fastcall FileRead(int Handle, void *Buffer, int Count);
```

其中 Handle 参数为文件句柄，由 FileCreate 或 FileOpen 函数得到；Buffer 参数指向读入数据的变量，Count 参数指定读取的字节数。

写文件函数 FileWrite 是这样定义的：

```
int __fastcall FileWrite(int Handle, const void *Buffer, int Count);
```

其中参数的意义与 FileRead 函数相同。

另外，读写文件操作一个必然需要的函数是 FileSeek 文件定位函数，FileSeek 函数如下定义：

```
int __fastcall FileSeek(int Handle, int Offset, int Origin);
```

其中 Handle 参数为定位文件句柄，Offset 参数告诉 FileSeek 函数定位点与原点（Origin）

之间的字节数，Origin 参数指定原点，它有三个值:

0	文件指针将移动到从文件起点算起 Offset 个字节的位置。
1	文件指针从文件当前指针点算起移动 Offset 个字节的位置。
2	文件指针从文件终点算起移动 Offset 个字节的位置。

当文件被建立或被打开时，文件指针在起点位置，利用 FileSeek 函数可以定量的移动文件指针到指定点。FileWrite 和 FileRead 函数都是从文件当前指针所指点开始读写的。

下面是一个具体的用到上面所讲函数的例子，它的作用是将一个文件打开读取，然后将每字节的内容换成 16 进制后，再写入另一文件。这个例子运行时如图 3-1 所示。

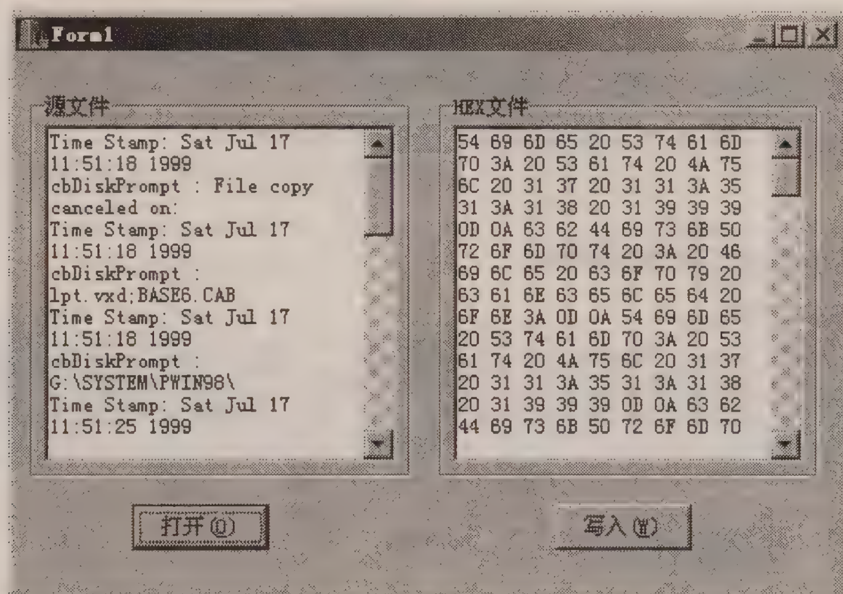


图 3-1 文件操作函数应用实例

这个程序的关键代码如下：

Source HexView 程序的关键代码

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    // 打开操作
    int iFileHandle;
    int iFileLength;
    int iBytesRead;
    char *pszBuffer;
    AnsiString hexBuffer;
    char hexchar[4];
    if (OpenDialog1->Execute())
    {
        Memo1->Clear();
        Memo2->Clear();
```

```
            // 打开文件
            iFileHandle = FileOpen(OpenDialog1->FileName, fmOpenRead);
            iFileLength = FileSeek(iFileHandle,0,2);// 获得数据长度
            FileSeek(iFileHandle,0,0);// 定位到起始点
            pszBuffer = new char[iFileLength+1];
            iBytesRead = FileRead(iFileHandle, pszBuffer, iFileLength);
            FileClose(iFileHandle);
            Memo1->Lines->LoadFromFile(OpenDialog1->FileName);
            // 化为 16 进制，并写入 Memo2 文本框
            for (int i=0;i<iBytesRead;i++)
            {
                sprintf(hexchar,"%02x ",int(pszBuffer[i]));
                hexBuffer+=AnsiString(hexchar).UpperCase();
            }
            Memo2->Lines->Add(hexBuffer);
            delete [] pszBuffer;
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{       // 写文件操作
        int iFileHandle;
        int iLength;
        if (SaveDialog1->Execute())
        {
            if (FileExists(SaveDialog1->FileName))
            {
                RenameFile(SaveDialog1->FileName,SaveDialog1->FileName+".BAK");
            }
            iFileHandle = FileCreate(SaveDialog1->FileName);
            for (int i=0;i<Memo2->Lines->Count;i++)
            {
                iLength = Memo2->Lines->Strings[i].Length();
                FileWrite(iFileHandle, Memo2->Lines->Strings[i].c_str(), iLength);
            }
            FileClose(iFileHandle);
        }
}
//---------------------------------------------------------------------------
```

Button1Click 函数进行读入工作。这里使用了 FileOpen，FileSeek 和 FileRead 函数，并利用

FileSeek 函数得到了文件的长度，从而确定了 FileRead 函数的 Count 参数的值。

写文件操作部分使用 FileExists 函数以确定用户提出的文件名是否已存在。如存在，则将原来文件进行备份，备份使用 RenameFile 函数对原始文件实施改名操作。

Memo2 框中的文本是被一行一行的写入文件的，该操作显示了文件指针在 FileWrite 函数执行后跟着移动。实际上，FileRead 函数也是如此。

从上例看来，C++ Builder 提供的文件操作函数是相当高效和易用的。所以，在 Windows 时代，使用标准的 C 输入输出函数进行文件操作已经毫无必要了。

3.1.3 用于文件操作的可视化组件

C++ Builder 提供一套用于目录或文件选择的可视化组件，包括驱动器组合框（TDriveComboBox）、目录列表框（TDirectoryListBox）、文件列表框（TFileListBox）等，这些组件都位于组件选项板的 Win 3.1 组件页，如图 3-2 所示。

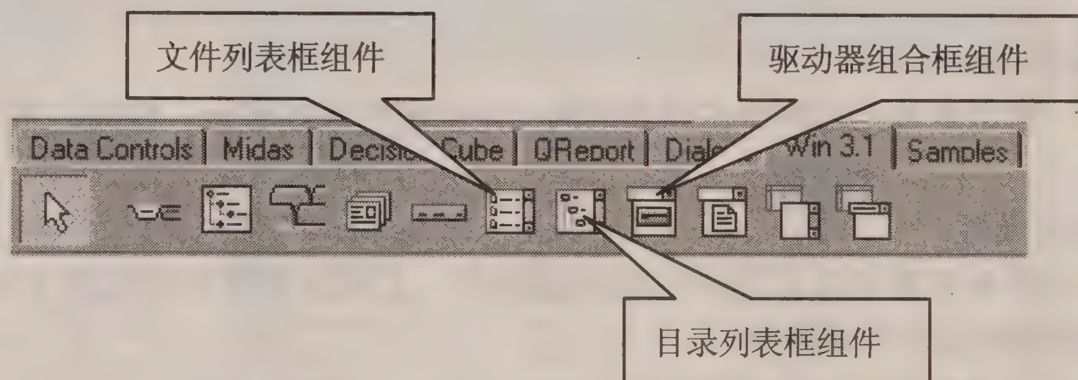


图 3-2 用于文件、目录、驱动器选择的组件

另外，C++ Builder 中还提供了一个 TCDirectoryOutline 目录树组件，这个组件位于选项板的 Samples 组件页。

1. 驱动器组合框（TDriveComboBox）

TDriveComboBox 用来实现驱动器组合框，用于以组合框的形式选择当前计算机的某个驱动器。它继承于 TComboBox 类，其主要属性有：

- DirList 属性。

目录列表。选择与本驱动器组合框相联系的目录列表框，当驱动器列表框有任何改变，例如用户选择了另一个驱动器，该变化能够使目录列表框作相应改变。

- Drive 属性。

字符串类型，用于设置和返回当前选择的驱动器名。

- TextCase 属性。

文本是否大小写，取值为 tcLowerCase 或 tcUpperCase。

2. 目录列表框（TDirectoryListBox）

TDirectoryListBox 用来实现一个目录列表框，显示某个驱动器上的目录树，并提供目录选择功能。它的主要属性有：

- Columns 属性。

设定列的数目。

- DirLabel 属性。

指定与本目录列表框相联系的静态文本组件。当目录列表框有任何变化，例如用户选了另一个目录时，能将任何变化送到相联系的静态文本组件，并将选择的目录显示出来。

- FileList 属性。

文件列表。选择与本目录列表框相联系的文件列表框，当目录列表框有任何改变，如用户选择了不同的目录时，该变化能够使文件列表框作用户选择的目录下的文件。

- Directory 属性。

字符串类型，用于设置和返回当前选择的目录名。

3. 文件列表框（TFileListBox）

TFileListBox 用来实现一个文件列表框，用于列出某一目录下的全部文件。它的主要属性有：

- ExtendedSelec 属性。

扩展选择。决定是否允许用户在列表框的一定范围内连续选择。例如，用户按下 Shift 键并用鼠标选择。

- FileEdit 属性。

指定与本文件列表框相联系的单行编辑框组件。文件列表框中所选的文件名，能够被及时的送到单行编辑框组件并显示出。

- FileName 属性。

当前被选文件的文件全名（包括目录）。

- FileType 属性。

文件类型。用于决定具有哪些属性的文件被显示在文件列表框中。可选值有：ftReadOnly、ftHidden、ftSystem、ftVolumeID、ftDirectory、ftArchive、ftNormal。

- Mask 属性。

可以根据通配符来决定在列表框所显示的文件，例如用“*.*”显示所有文件。

- MultiSelect 属性。

多选属性，用于设置是否允许选择多个文件。

下面是一个简单的文件操作组件的演示程序（如图 3-3 所示），它演示了上面讲述的各个组件的联合使用，当用户单击 FileListBox 框时，用户所选的文件全名将被显示在右边的列表框内。

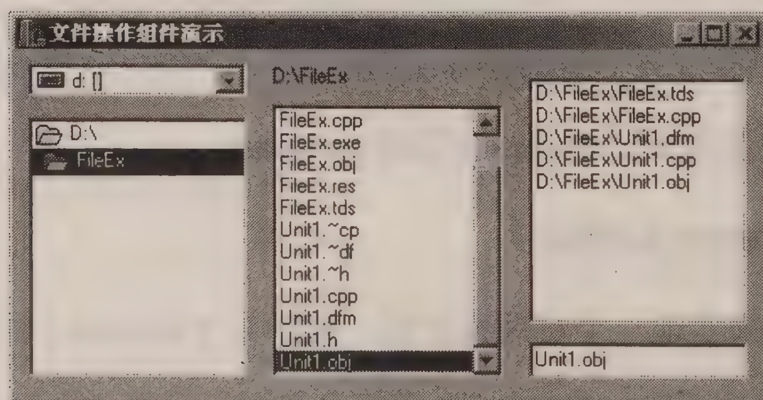


图 3-3 文件操作组件演示程序

C++ Builder 中的另一个用于文件和目录可视化操作的组件是位于 Samples 页的 TCDirectoryOutline 目录树组件。该组件显示一个包含驱动器和目录的树形图，它的作用与 TDirectoryListBox 目录列表框类似，但它同时支持了驱动器的选择。

3.2 实例创建分析

本章的目标是完成一个类似 Windows 95 下的资源管理器的“BCB 文件管理器”程序。文件管理器范例程序的主要功能是文件信息检视和文件处理（包括拷贝、删除、剪切等等），如图 3-4 所示。

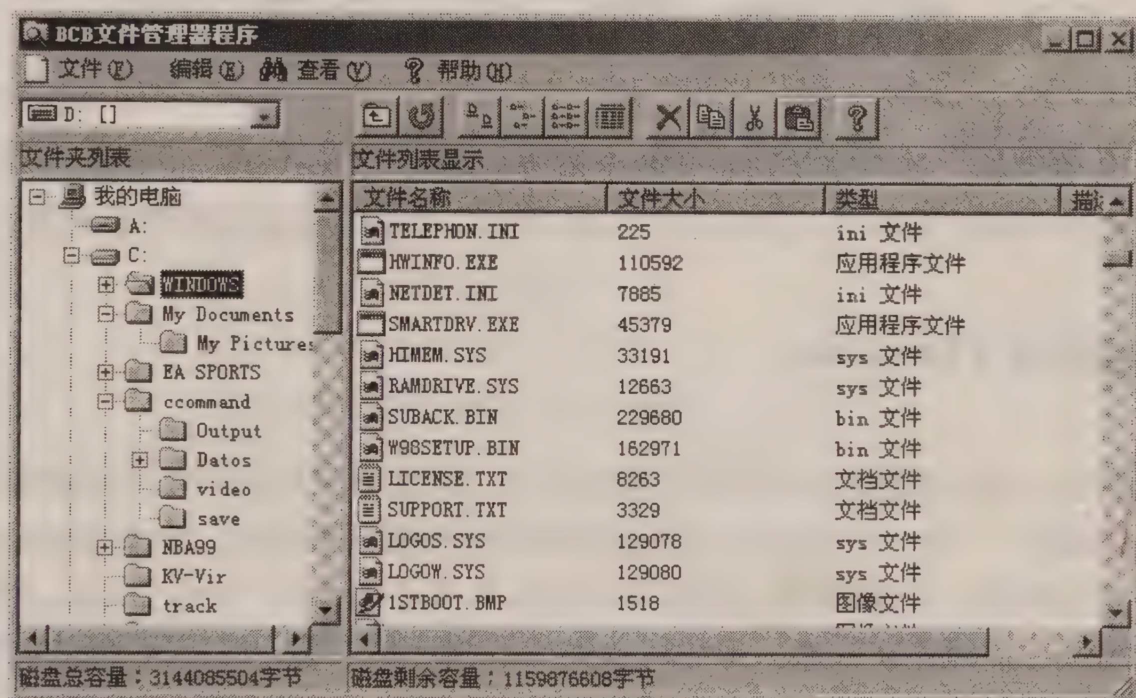


图 3-4 BCB 文件管理器程序

应用 3.1 节讲述的文件操作组件，可以快速完成一个包括上述功能的文件管理器程序。但这样实现的程序将是非常“简陋”的。这是因为，C++ Builder 提供的几个文件和目录的组件基于 Windows 3.X 风格，以现在的眼光来看，新的 Win32 风格的界面更加美观大方和具有亲和力。

本章的目标是实现 Win32 风格的文件管理器程序，该程序以 Windows 95 资源管理器为蓝本。Windows 95 资源管理器为人们所熟知：总体来看其界面可分为两大部分，左边是包含用户计算机目录树的目录选择框，右边是一个文件列表框。在 C++ Builder 中提供了同样风格的组件，分别是 TTreeView 树视图组件和 TListView 列表视图组件。我们将使用这两个组件实现 BCB 文件管理器程序。

实现该程序的主要任务分为两点：文件的浏览功能和文件处理功能。具体来说，在本章中将逐步完成以下部分：

- 查找磁盘上的目录和文件。
- 树视图和列表视图显示文件和目录的实现。
- 与文件浏览相关命令的响应。
- 当用户双击文时，执行或打开文件。

- 文件复制、剪切件删除等功能。
- 显示和更改文件属性功能。

BCBEditor 工程中主要包含下面的一些文件：

- 主窗体单元。类名：TForm1，文件 Unit1.cpp、Unit1.h。
- 用于显示和更改文件属性的窗体单元。类名：TForm2，文件：Unit2.cpp、Unit2.h。
- About 窗体。类名：TAboutBox1，文件：About.cpp，About.h。

BCB 文件管理器范例用于演示文件操作、文件查找功能编程以及 Win32 风格的 TTreeView 树视图组件和 TListView 列表视图组件的使用。相信通过这个实例将使你学到很多 C++ Builder 中文件处理方面的技术和技巧。

3.3 界面风格：TTreeView 和 TListView

"BCB 文件管理器"的主窗体用到了两个重要的组件 TTreeView 和 TListView，这两个组件都位于选项板的 Win32 组件页，分别用于实现 Win32 风格的树视图和列表视图。

3.3.1 树视图组件 TTreeView

树视图组件是一个以树状结构显示信息的组件。树视图中形成树状结构的基本单位称为树节点（TTreeNode），每一个节点可以通过文本和一个图标来标识，一个节点还可以包含子节点。一个树型结构有一个根节点，除根节点外每个节点都存在一个父节点，根节点、父节点、子节点之间通过虚线连在一起，形成树状结构，并反映了事物的层次和结构关系。图 3-5 显示了树视图组件。



图 3-5　树视图组件

以下介绍树视图组件的重要的属性、事件和方法。

1. 树视图组件的重要属性

- AutoExpand 属性。

决定树视图中的节点是否在被选中时自动展开。

- Indent 属性。

用于设定子节点展开时和父节点之间的距离，单位为像素。

- Images 属性。

声明：__property TCustomImageList* Images = {read=FImages, write=SetImages};

用于设定与树视图相联系的图标列表（ImageList），树视图节点项的图标可由 Images 指

向的图标列表内的图标提供。

- Items 属性。

声明：__property TTreeNodes* Items = {read=FTreeNodes, write = SetTreeNodes};

指向树视图节点列表类（TTreeNodes），此属性包含了树视图中所有节点项。

- Selected 属性。

树视图节点类类型，描述树视图中被选中的节点项。

- ShowButtons 属性。

用于设定是否在树视图的父节点左侧显示+或-符号，以表示树的节点是否展开。

- ShowLines 属性。

用于设定是否显示各个节点之间的连线。

- ShowRoot 属性。

决定树视图是否显示根节点项。

- SortType 属性。

此属性用于确定树视图各个节点的排序方式，为 TSortType 类型，有下面四种可能取值：

stNone	不进行排序操作。
stData	按照节点项的数据进行排序。
stText	按照节点项的标题属性进行排序。
stBoth	同时考虑数据和标题属性两个因素。

- ToolTips。

决定树视图中的节点是否显示暗示文本(Hints)。

- TopItem。

当前树视图框中最上面的节点，树视图节点类类型。设置这个属性可以使树视图框滚动到指定位置显示。

2. 树视图组件的重要方法

树视图组件的重要方法有：

- AlphaSort 方法。

用于对树视图节点进行排序操作。

- CustomSort 方法。

声明：typedef int (CALLBACK *PFNTVCOMPARE)(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort);

bool __fastcall CustomSort (PFNTVCOMPARE SortProc, int Data);

树视图组件支持用户定义的排序方法，使用 CustomSort 方法需要首先定义一个排序函数。

- FullCollapse/FullExpand 方法。

使树视图组件收缩/展开所有的节点项。

- GetNodeAt 方法。

声明：TTreeNode* __fastcall GetNodeAt (int X, int Y)

它的功能是获取树视图框内 X、Y 坐标处的节点项。

- IsEditing 方法。

用于确定树视图中某一的节点项是否正在被用户编辑。

- LoadFromFile/LoadFromStream/SaveToFile/SaveToStream 方法。

树视图支持的存取函数。

3. 节点列表类 TTreeNodes

在树视图中，树型结构是由许多节点构成的，所以树视图节点类（TTreeNodes）就显得非常重要。

在树视图类中，所有节点又组成了一个节点列表类（TTreeNodes），树视图节点列表类实现以链表结构存储和操作树视图节点。

TTreeView 类的 Items 属性是一个节点列表类的实例，通过该属性可以访问一个树视图的所有节点项。此属性这样声明：

__property TTreeNode* Item[int Index] = {read=GetNodeFromIndex};

可见通过这个属性，可以使用 Index 索引值来顺序访问树结构的各个节点，同时也可以通过 Count 属性访问目前节点的总数。

TTreeNodes 类的重要方法说明如下：

- Add 方法。

声明：TTreeNode* __fastcall Add(TTreeNode* Node, const System::AnsiString S);

Add 方法用于在指定节点项处增加兄弟节点。

Node 参数指明节点项，参数 S 指定增加节点的文本，此方法返回增加的节点对象。

- AddChild 方法。

声明：TTreeNode* __fastcall AddChild(TTreeNode* Node, const System::AnsiString S);

AddChild 方法用于在指定节点处增加其子节点，此方法用法与 Add 类似。

- Clear 方法。

用于清除所有节点项。

- Delete 方法。

声明：void __fastcall Delete(TTreeNode* Node);

用于删除指定节点项。

- GetFirstNode 方法。

声明：TTreeNode* __fastcall GetFirstNode (void);

用于获得节点列表对象的第一个节点项。

- GetNode 方法。

声明：TTreeNode* __fastcall GetNode(HTREEITEM ItemId);

通过节点类的句柄，获得该句柄的节点项。

4. 树视图节点类 TTreeNode

通过节点列表类（TListNode）可以访问每一个节点，而每个节点都是一个树视图节点类（TTreeNode）类型。

TListNode 有三个属性最为重要：Text 属性，节点类的文本标示，决定节点显示出的文本内容；ImageIndex 属性，指明该节点项所用的图标在图表列表中的位置；Index 属性，索引值，表明了节点项在兄弟节点中的索引。

树视图节点类的重要方法包括：

- EditText 方法。

用于使树视图中节点处于编辑状态，并使编辑部分获得焦点，使得用户可以进行文本的编辑。

- Delete 方法。

删除自身节点项。

- DeleteChildren 方法。

删除自身所有子节点项。

- Collapse/ Expanded 方法。

收缩/展开本节点。

3.3.2 列表视图组件

列表视图组件是一个可用多种方式（大图标、小图标、列表和报告方式）来显示信息单元的组件。列表视图组件的基本显示单元是列表项，列表项可由文本标识和图标共同组成，列表项还可以包含其他文本信息（报告方式时显示），列表视图组件如图 3-6 所示。



图 3-6 列表视图组件

下面具体介绍列表视图组件的重要属性：

- Checkboxes 属性。

决定列表视图是否支持检查框。

- GridLines 属性。

是否使用网格线，即使该属性为真，网格线也只能以报告方式显示出来。

- ItemFocused 属性。

列表项 TListItem 类型，其值为当前获得焦点的列表项。

- Items 属性。

TListItems 类型。它表示所有列表项的总和，可以通过此属性访问所有的列表项。

- MultiSelect 属性。

是否允许多选。

- Selected 属性。

列表项 TListItem 类型，其值为当前选中的列表项。

- ViewStyle 属性。

为 TViewStyle 枚举类型，其值可以是：

vsIcon	以大图标的方式显示列表项。
vsSmallIcon	以小图标的方式显示列表项。
vsList	按列以小图标方式显示列表项。

vsReport　　以报告方式显示，可以显示列表项的其他信息。

列表视图组件也包含很多重要方法，包括：

- Arrange 方法。

当列表视图以大图标或小图标的方式显示时，可以使用此方法队列表项重新排列。

- CustomSort 方法。

列表视图组件支持用户定义的排序方法。类似 TTreeView 组件的 CustomSort 方法，使用该方法需要首先定义一个排序函数。

- FindCaption 方法。

声明：TListItem* __fastcall FindCaption(int StartIndex, System::AnsiString Value, bool Partial, bool Inclusive, bool Wrap);

用于搜索拥有特定文本名称的列表项。

- GetItemAt 方法。

声明：TListItem* __fastcall GetItemAt(int X, int Y);

此方法用于获取指定位置处的列表项。

- Scroll 方法。

声明：void __fastcall UpdateItems(int FirstIndex, int LastIndex);

此方法用于列表视图框的滚动操作。

- UpdateItems 方法。

声明：void __fastcall UpdateItems(int FirstIndex, int LastIndex);

用于刷新指定范围的列表视图窗口。

3.3.3 创建范例程序界面

如前文所述，构成“BCB 文件管理器程序”的主窗体界面的主要组件是一个 TTreeView 组件和一个 TListView 组件，如图 3-7 所示。

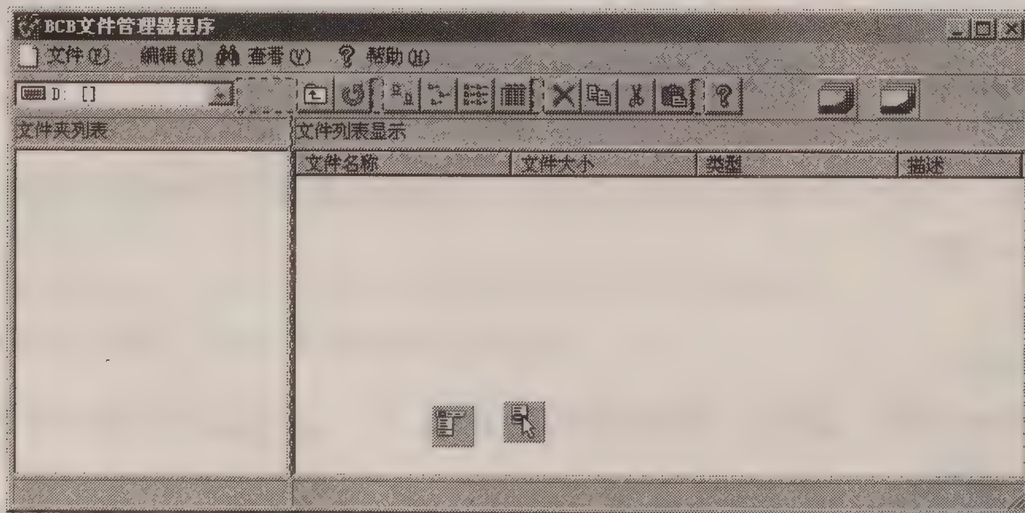


图 3-7　主窗体界面设计

主窗体中包含了一个工具栏（TToolBar），工具栏上设置一个驱动器组合框和 11 个工具按钮。主窗体中部，放置两个大的面板（TPanel）组件，两面板之间放置一个 TSplitter 分隔条

组件。注意，第一个面板的 Align 属性应置为 alLeft，第二个面板组件 Align 属性应设为 alClient。再分别在这两个面板上方放置另外两个面板组件，用于显示信息。这两个面板组件的 Align 属性应设为 alTop，并控制其大小至合适。而后在两个大面板上分别放置一个 TTreeView 树视图组件 和 TListView 列表视图组件，它们的 Align 属性都设为 alClient。

再后可以对 TListView 列表视图组件等组件的属性设置初始值，例如为 TListView 列表视图组件增加几个栏，并设置 ViewStyle 为 vsReport 等等。

最后可以编辑或建立菜单、状态栏和一些非可视组件，主要是两个 ImageList 图像列表组件，分别用于菜单及工具栏按钮的图标和用于 TTreeView 树视图组件及 TListView 列表视图组件的图标。

应该注意以上界面设置中面板组件和 Align 属性的运用，这样设置可以使窗体大小改变或最大化时显示正确。

3.4 文件管理和浏览

文件管理器的首要功能是使用户可以通过树形图和列表视图方便地浏览计算机上的所有文件和目录。完成这个目的需要两个步骤：

(1) 搜索计算机上的文件和目录，以获得必要的信息。

(2) 将找到的文件和目录在树视图和列表视图中组织并显示出来。

其实在本程序中这两部分是同时执行的，即一边获取需要的文件和目录，一边将文件目录的信息加入树视图或列表视图。下面将具体介绍文件浏览功能的实现方法。

3.4.1 初始化工作

由于在列表视图中只显示某一个目录下的文件和目录信息，需要在主窗体头文件中定义一个变量指明这个目录。

```
AnsiString CurrentDir;
```

然后进行初始化工作，在主窗体的 OnCreate 响应函数如下：

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    CurrentDir=GetCurrentDir();
    UpdateTreeView();
    UpdateListView();
    ComState=csNone;
}
```

GetCurrentDir 用于获得系统的当前目录。而后可以进行树视图 TreeView1 的创建工作。最后一行代码用于文件操作，描述当前操作命令组状态。它是程序中定义的 TCommandState 枚举类型变量，在头文件中如下定义：

```
enum TCommandState { csCopy, csCut, csNone};
```

该变量与文件操作相关，在本章稍后还将作具体说明。

3.4.2 树视图的组织和显示

在函数 UpdateTreeView 中实现了树视图的生成和显示。这个函数如下：

Source 树形视图的生成和显示

```
//---------------------------------------------------------------------------
void TForm1::UpdateTreeView()
{
        TTreeNode *rNode,*mNode;
        TreeView1->Items->Clear();
        rNode=TreeView1->Items->Add(TreeView1->Selected,"我的电脑"); // 加入根节点
        rNode->ImageIndex=0;
        rNode->SelectedIndex=0;
        mNode=TreeView1->Items->AddChild(rNode,"A:");
        mNode->ImageIndex=4;
        mNode->SelectedIndex=4;
        AddDirectory("A:",mNode); // 加入软驱目录
        for (int i=1;i<Drive1->Items->Count;i++)
        {
                mNode=TreeView1->Items->AddChild (rNode,AnsiString(char('B'+i))+":");
                mNode->ImageIndex=3;
                mNode->SelectedIndex=3;
                AddDirectory(AnsiString(char('B'+i))+":",mNode);
        }
}
//---------------------------------------------------------------------------
```

在此函数中，首先是加入一个根节点“我的电脑”。需要对 SeletedIndex 属性进行设置，使得在选择时图标与通常图标保持一致。然后加入第一层节点，即各个驱动器，这里假设用户只有一个软驱 A：，其他硬盘和光盘从 C：开始。

这里通过驱动器组合框来获得可用驱动器的数目，并一一加入树形图中。对于每个驱动器节点用 AddDirectory 函数为其添加目录子节点。AddDirectory 函数如下：

Source 实现 AddDirectory 函数

```
//---------------------------------------------------------------------------
void TForm1::AddDirectory(AnsiString path,TTreeNode *fNode)
```

```
{
	TSearchRec sr;
	TTreeNode *mNode;
	if (FindFirst(path+"\\*.*",faDirectory,sr)==0){
		if (sr.Attr==faDirectory){
			if (sr.Name!="."&&sr.Name!=".."){
				mNode=TreeView1->Items->AddChild(fNode,sr.Name);
				mNode->ImageIndex=1;
				mNode->SelectedIndex=2;
				AddDirectory(path+"\\"+sr.Name,mNode);
			}
		}
	}
	while (FindNext(sr)==0){
		If (sr.Attr==faDirectory){
			if (sr.Name!="."&&sr.Name!=".."){
				mNode=TreeView1->Items->AddChild(fNode,sr.Name);
				mNode->ImageIndex=1;
				mNode->SelectedIndex=2;
				AddDirectory(path+"\\"+sr.Name,mNode);
			}
		}
	}
	FindClose(sr);
}
//---------------------------------------------------------------------------
```

这里用到了一组非常重要的函数：FindFirst、FindNext 和 FindClose，这组函数用于文件或目录的查找。在使用这组函数时，必须定义一个 TSearchRec 结构的变量，用于返回查找结果，这个结构的定义为：

```
struct TSearchRec
{
	int Time;
	int Size;
	int Attr;
	AnsiString Name;
	int ExcludeAttr;
	int FindHandle;
	_WIN32_FIND_DATAA FindData;
};
```

可以看到，它包含了程序员需要关心的所有文件属性。

FindFirst 是一组查找工作的开始函数，它用于指定查找路径和查找文件类型，此函数的原型如下：

int __fastcall FindFirst(const AnsiString Path, int Attr, TSearchRec &F);

其中 Attr 参数用于指明查找文件类型，它可取值，如图 3-1 所示，并可使用“|”按位或来选择多个类型。

表 3-1　文件属性常量及其含义

常 量	数 值	含 义
faReadOnly	0x00000001	只读文件
faHidden	0x00000002	隐藏文件
faSysFile	0x00000004	系统文件
faVolumeID	0x00000008	磁盘卷标文件
faDirectory	0x00000010	目录文件
FaArchive	0x00000020	归档文件
FaAnyFile	0x0000003F	任何文件

FindFirst、FindNext、FindClose 函数一般的使用模式是：

FindFirst —〉 FindNext —〉 FindNext —〉 ······ —〉 FindNext —〉 FindClose ;

这里需要注意两点：第一，FindNext 为 FindFirst 函数的延续，查找路径和查找文件类型应与 FindFirst 相同；第二，查找完毕后务必使用 FindClose 函数释放内存资源。

3.4.3　列表视图的组织和显示

在主窗体的 OnCreate 响应函数中，我们调用 UpdateListView 函数用于创建和显示列表视图。创建列表视图包括两部分：搜索当前目录的子目录并予显示；搜索当前目录下的所有文件并予显示。这两部分工作分别在 ListViewAddDirectory、ListViewAddFile 函数中实现，并在 UpdateListView 函数中调用，以下是这部分的代码：

Source　列表视图的组织和显示

```
//---------------------------------------------------------------------------
void TForm1::UpdateListView()
{
        ListView1->Items->Clear();
        ListViewAddDirectory();
        ListViewAddFile();
}
//---------------------------------------------------------------------------
void TForm1::ListViewAddDirectory()
{
        // 搜索子目录并显示
```

```
    TListItem *mItem;
    TSearchRec sr;
    if (FindFirst(CurrentDir+"\\*.*",faDirectory,sr)==0){
        if (sr.Attr==faDirectory&&sr.Name!="."&&sr.Name!=".."){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=1;
                mItem->SubItems->Add("");
                mItem->SubItems->Add("文件目录");
                mItem->SubItems->Add("");
        }
    }
    while (FindNext(sr)==0){
        if (sr.Attr==faDirectory&&sr.Name!="."&&sr.Name!=".."){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=1;
                mItem->SubItems->Add("");
                mItem->SubItems->Add("文件目录");
                mItem->SubItems->Add("");
        }
    }
    FindClose(sr);
}
//---------------------------------------------------------------------------
void TForm1::ListViewAddFile()
{   // 搜索文件并显示
    TListItem *mItem;
    TSearchRec sr;
    AnsiString fileext;
    if (FindFirst(CurrentDir+"\\*.*",0,sr)==0){
        if (sr.Attr!=faDirectory){
            fileext=ExtractFileExt(sr.Name).LowerCase();
            if (fileext==".exe"){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=7;
                mItem->SubItems->Add(AnsiString(sr.Size));
                mItem->SubItems->Add("应用程序文件");
                mItem->SubItems->Add("");
```

```
            }
            else if(fileext==".txt"||fileext==".doc"){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=6;
                mItem->SubItems->Add(AnsiString(sr.Size));
                mItem->SubItems->Add("文档文件");
                mItem->SubItems->Add("");
            }
            else if(fileext==".bmp"||fileext==".jpg"||fileext==".gif"){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=8;
                mItem->SubItems->Add(AnsiString(sr.Size));
                mItem->SubItems->Add("图像文件");
                mItem->SubItems->Add("");
            }
            else{
                fileext.Delete(1,1);
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=5;
                mItem->SubItems->Add(AnsiString(sr.Size));
                mItem->SubItems->Add(fileext+" 文件");
                mItem->SubItems->Add("");
            }
        }
        while (FindNext(sr)==0){
        if (sr.Attr!=faDirectory){
            fileext=ExtractFileExt(sr.Name).LowerCase();
            if (fileext==".exe"){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=7;
                mItem->SubItems->Add(AnsiString(sr.Size));
                mItem->SubItems->Add("应用程序文件");
                mItem->SubItems->Add("");
            }
            else if(fileext==".txt"||fileext==".doc"){
                mItem=ListView1->Items->Add();
```

```
                    mItem->Caption=sr.Name;
                    mItem->ImageIndex=6;
                    mItem->SubItems->Add(AnsiString(sr.Size));
                    mItem->SubItems->Add("文档文件");
                    mItem->SubItems->Add("");
                }
                else if(fileext==".bmp"||fileext==".jpg"||fileext==".gif"){
                    mItem=ListView1->Items->Add();
                    mItem->Caption=sr.Name;
                    mItem->ImageIndex=8;
                    mItem->SubItems->Add(AnsiString(sr.Size));
                    mItem->SubItems->Add("图像文件");
                    mItem->SubItems->Add("");
                }
                else{
                    fileext.Delete(1,1);
                    mItem=ListView1->Items->Add();
                    mItem->Caption=sr.Name;
                    mItem->ImageIndex=5;
                    mItem->SubItems->Add(AnsiString(sr.Size));
                    mItem->SubItems->Add(fileext+" 文件");
                    mItem->SubItems->Add("");
                }
            }
        }
    }
    FindClose(sr);
}
//---------------------------------------------------------------------------
```

在 ListViewAddFile 函数中通过调用 FindFirst、FindNext 函数搜索指定目录下的所有文件，并使用 C++ Builder 的文件操作函数 ExtractFileExt 确定文件扩展名，然后根据文件的扩展名使用相应的图标给予显示和填写相应的属性信息。

在完成列表视图之后，就可以给用户提供四种不同的方式浏览，这不需要更多的代码，改变 ListView1 的 ViewStyle 属性就可以达到此目的，四种不同的浏览方式图标如图 3-8 所示。

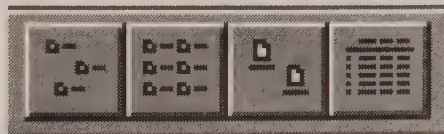


图 3-8 四种不同的浏览方式

这里只需注意工具栏按钮必须和相应菜单的选项状态保持一致。仅举一例：

```
void __fastcall TForm1::ToolButton3Click(TObject *Sender)
{
        ListView1->ViewStyle=vsSmallIcon;
        UpdateMenu();
}
void TForm1::UpdateMenu()
{
        N1->Checked=ListView1->ViewStyle==vsIcon;
        N2->Checked=ListView1->ViewStyle==vsSmallIcon;
        N3->Checked=ListView1->ViewStyle==vsList;
        N4->Checked=ListView1->ViewStyle==vsReport;
}
```

3.4.4 用户浏览命令的响应

在本程序中，用户可通过四个手段改变当前的浏览状态：改变驱动器组合框；单击树视图组件所显示的相应目录；单击列表视图中显示的相应目录；单击工具栏上提供的“上一级目录”按钮。下面将分别阐述其实现办法。

对于驱动器组合框来说，应该响应它的 OnChange 事件，代码如下：

```
void __fastcall TForm1::Drive1Change(TObject *Sender)
{
        CurrentDir=Drive1->Drive;
        CurrentDir+=":";
        UpdateListView();
        DWORD Total,Free;
        Total=DiskSize(Drive1->Drive-'A'+1);
        Free=DiskFree(Drive1->Drive-'A'+1);
        StatusBar1->Panels->Items[0]->Text=
                "磁盘总容量："+AnsiString(Total)+"字节";
        StatusBar1->Panels->Items[1]->Text=
                "磁盘剩余容量："+AnsiString(Free)+"字节";
}
```

当驱动器组合框改变时，将使列表视图显示相应驱动器的根目录状态。同时还将显示当前驱动器的总容量和剩余容量，这里用到了 DiskSize 和 DiskFree 函数，这两个函数的参数为整型变量，1 代表 A 驱，2 代表 B 驱，依次类推。

对于树视图来说，应响应其 OnClick 事件。

```
void __fastcall TForm1::TreeView1Click(TObject *Sender)
{
```

```
        AnsiString dName[20],dir;
        int i=0,j;
        TTreeNode *mNode=TreeView1->Selected;
        if (mNode->Text!="我的电脑"){
            if (mNode!=NULL){
            do{
                dName[i]=mNode->Text;
                mNode=mNode->Parent;
                i++;
            } while (mNode->Text!="我的电脑");
            dir=dName[i-1];
            for (j=i-2;j>=0;j--)
                dir=dir+"\\"+dName[j];
                CurrentDir=dir;
            }
            UpdateListView();
        }
    }
```

这段代码使用逆推的方法找出单击的目录的全路径，关键利用了 TTreeNode 类的 Parent 属性，获取全路径后将该目录内容显示在列表视图框中。

列表视图响应 OnDblClick 事件，此响应函数与菜单的“打开”项 OnClick 事件共用。

```
void __fastcall TForm1::ListView1DblClick(TObject *Sender)
{
    if (ListView1->SelCount==1)
    {
        if (ListView1->Selected->ImageIndex==1) {
            CurrentDir+="\\"+ListView1->Selected->Caption;
            UpdateListView();
        }
        else {
            AnsiString FileStr;
            FileStr=CurrentDir+"\\"+ListView1->Selected->Caption;
            ShellExecute(Handle,"open",FileStr.c_str(),
                NULL,NULL,SW_SHOWNORMAL);
        }
    }
}
```

首先通过 ImageIndex 属性判断是否双击了目录，若是，则改变 CurrentDir，并将目录内容显示出来。如果是文件，则使用 ShellExcute API 函数打开它。这个函数是这样定义的：

```
HINSTANCE ShellExecute(
```

```
        HWND hwnd,
        LPCTSTR lpOperation,
        LPCTSTR lpFile,
        LPCTSTR lpParameters,
        LPCTSTR lpDirectory,
        INT nShowCmd
);
```

此函数的功能与在 Windows 95/98 中里双击某个文件效果一致。该函数会自动找到 lpFile 参数指定文件的默认打开程序，并执行该程序使 lpFile 参数指定的文件打开。例如，当用户在列表视图中双击了*.dfm 的文件，C++ Builder 4 将被执行并打开双击的那个窗体定义文件。lpOperation 参数设定操作命名，有三个值：open、print、explore。若使用 print，则 lpFile 指定文件应为文档文件；若使用 explore，则指定文件应为文件夹。

nShowCmd 参数为打开时的显示状态，其中几个重要的值是：

SW_SHOWNORMAL	普通显示状态。
SW_SHOWMAXIMIZED	打开并显示为最大化。
SW_SHOWMINIMIZED	打开并显示为最小化。
SW_HIDE	隐藏窗口。

最后实现“上一级目录”按钮的功能。

```
void __fastcall TForm1::ToolButton14Click(TObject *Sender)
{
    int i=CurrentDir.Length();
    while (i>0){
        if (CurrentDir[i]=='\\'){
            CurrentDir.Delete(i,CurrentDir.Length()-i+1);
            UpdateListView();
        break;
        }
        i--;
    }
}
```

这里所做的是找到 CurrentDir 的最后一个“\”字符，并将此字符连同字符后的所有字符删去，这样就得到了上一级目录的路径字符串，而后执行刷新列表视图。

3.5 实现文件操作功能

本范例程序中实现了文件的拷贝、剪切、删除、重命名和文件属性的检视及修改，这些功能相对来说较容易实现。

3.5.1 文件的拷贝、剪切、删除

程序中实现类似 Windows 95 资源管理器的拷贝或剪切方式，即当用户发出拷贝命令时，并未直接进行拷贝操作，而是将待拷文件的信息放入剪贴板（在本程序中，并没有真的放入剪贴板，而是模拟了剪贴板功能），等到用户在适合的目录下发出粘贴命令时才真的进行了复制。文件操作工具按钮如图 3-9 所示。



图 3-9　文件操作工具按钮

当用户发出拷贝或剪切命令时，用变量将目标文件记录下来。在头文件定义两个变量：

```
AnsiString TargetFile; // 目标文件全名
AnsiString TargetFN;  // 目标文件名，相当 ExtractFileName(TargetFile);
```

以下是实现拷贝、剪切、粘贴和删除功能的代码：

Source 实现文件拷贝、剪切、粘贴和删除功能

```
//---------------------------------------------------------------------------
void __fastcall TForm1::ToolButton8Click(TObject *Sender)
{
        // 拷贝命令响应函数
        if (ListView1->SelCount==1){
                TargetFile=CurrentDir+"\\"+
                ListView1->Selected->Caption;
                TargetFN=ListView1->Selected->Caption;
                ComState=csCopy;
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::ToolButton9Click(TObject *Sender)
{
        // 剪切命令响应函数
        if (ListView1->SelCount==1){
                TargetFile=CurrentDir+"\\"+
                ListView1->Selected->Caption;
                TargetFN=ListView1->Selected->Caption;
                ComState=csCut;
```

```
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::ToolButton13Click(TObject *Sender)
{
        // 粘贴命令响应函数，分两种情况处理
        if (ComState==csCopy){
                if (MessageBox(Handle,("真的要将"+TargetFile+"文件拷贝到"
                        +CurrentDir+"目录下吗?").c_str(),
                        "拷贝确认",MB_OKCANCEL|MB_ICONQUESTION)==IDOK){
                        CopyFile(TargetFile.c_str(),
                                (CurrentDir+"\\"+TargetFN).c_str(),false);
                        UpdateListView();
                }
}
        else if (ComState==csCut){
                if (MessageBox(Handle,("真的要将"+TargetFile+"文件移动到"
                        +CurrentDir+"目录下吗?").c_str(),
                        "剪切确认",MB_OKCANCEL|MB_ICONQUESTION)==IDOK){
                        MoveFile(TargetFile.c_str(),
                                (CurrentDir+"\\"+TargetFN).c_str());
                        TargetFile=CurrentDir+"\\"+TargetFN;
                        ComState=csCopy;
                        UpdateListView();
                }
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::ToolButton7Click(TObject *Sender)
{
        // 删除操作函数
        if (ListView1->SelCount==1){
                AnsiString FileName;
                FileName=CurrentDir+"\\"+ListView1->Selected->Caption;
                if (MessageBox(Handle,("真的要删除"+FileName+"文件吗?").c_str(),
                        "删除确认",MB_OKCANCEL|MB_ICONWARNING)==IDOK){
                        DeleteFile(FileName.c_str());
                        UpdateListView();
                }
        }
```

```
}
//---------------------------------------------------------------------------
```

当然，完全可以使用 FileWrite 和 FileRead 函数编写一个拷贝函数进行拷贝操作。但范例程序在这里使用了另一种方法，即利用 API 函数 CopyFile，此函数用法简单，它的声明如下：

```
BOOL CopyFile(
  LPCTSTR lpExistingFileName, // 指向源文件名的字符串
  LPCTSTR lpNewFileName,     // 指向目标文件名的字符串
  BOOL bFailIfExists         // 目标文件已存在时的操作标志
 );
```

该函数需要传递源文件名和目标文件名的字符串，同时需要给出一个目标文件已存在时的操作标志 bFailIfExists。如果该值为 false，该函数将覆盖已存在的目标文件；否则，如果目标文件存在，拷贝操作将失败。

这个函数包含在 ShellApi.h 头文件中。

3.5.2 Win32 风格文件重命名的实现

在早期的 Windows3.X 的文件管理器（FileManager）中，修改文件是通过一个对话框实现的。而 Windows 95 的资源管理器所提供的文件重命名方式，可以直接在列表视图中通过编辑文件名进行，不仅直观，而且快捷有效，可以说是一大进步。在 BCB 文件管理器范例程序中，将实现这种 Win32 风格的文件改名操作。Win32 风格的文件改名操作如图 3-10 所示。

文件名称	文件大小	类型
DBLBUFF.SYS	2614	sys 文件
IFSHLP.SYS	3708	sys 文件
CONFIG.TXT	11478	文档文件
DISPLAY.TXT	13742	文档文件
FAQ.TXT	8888	文档文件
GBK.TXT	110421	文档文件

图 3-10　Win32 风格的文件改名操作

实现这项功能的关键在于三点：TListView 组件的 OnEdited 事件，OnEditing 事件和列表项 TListItem 的 EditCaption 方法。

用户有两种方式进行重命名操作：其一，对于在列表视图中已选中的列表项再次单击，此列表项将进入名字的编辑状态，用户编辑结束时，确认并完成改名操作；其二，通过菜单或者弹出菜单中的“重命名”项进入文件名编辑状态，完成改名。无论哪种方式，关键都在于对 OnEdited 事件和 OnEditing 事件的响应处理。

首先，当进入文件名编辑状态时，OnEditing 事件被触发，在此需要记录下修改文件的原名，以保证已获取改名操作的对象文件名。

```
void __fastcall TForm1::ListView1Editing(TObject *Sender, TListItem *Item, bool &AllowEdit)
```

```
{
    RenameFN=ListView1->Selected->Caption;
}
```

然后，当用户完成文件名编辑时，OnEdited 事件被触发，此时可以获得用户输入的新文件名，并进行重命名操作。

```
void __fastcall TForm1::ListView1Edited(TObject *Sender, TListItem *Item, AnsiString &S)
{
    if (MessageBox(Handle,("真的要将"+RenameFN+"文件重命名为"+S+"吗?").c_str(),
        "重命名确认",MB_OKCANCEL|MB_ICONQUESTION)==IDOK){
        RenameFile((CurrentDir+"\\"+RenameFN).c_str(),
            (CurrentDir+"\\"+S).c_str());
    }
    else UpdateListView();
}
```

此处使用了 C++ Builder 提供的 RenameFile 函数进行改名操作。如果用户取消操作，则刷新列表视图以显示正确的文件名。

如果是由菜单项进入文件名编辑，只需调用 TListItem 的 EditCaption 方法。

```
void __fastcall TForm1::N11Click(TObject *Sender)
{
    ListView1->Selected->EditCaption();
}
```

3.5.3 文件属性的检视与修改

关于文件属性的操作没有什么特别复杂的东西，主要是利用 C++ Builder 提供的 FileGetAttr、FileSetAttr 和 FileAge 函数。当然，具体文件属性操作实现还存在一些涉及时间转换和属性提取的问题。

首先我们需要为文件属性操作制作一个窗体，如图 3-11 所示。

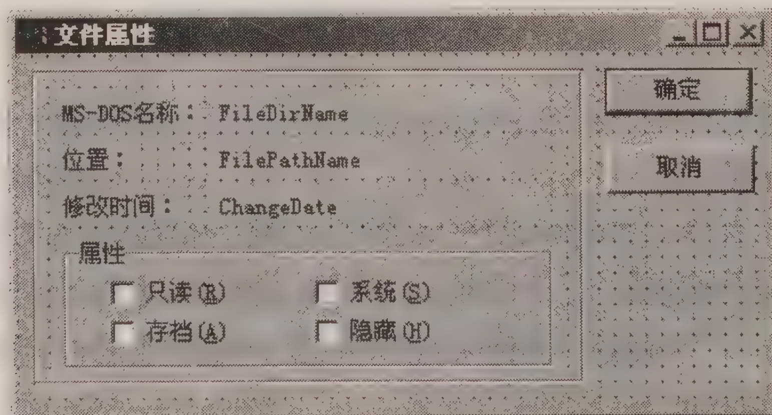


图 3-11　用于文件属性操作的窗体

该窗体主要是一些 Label 和 CheckBox 组件，以下是进行文件属性检视和修改的代码：

Source 查看文件属性功能的实现

```
//---------------------------------------------------------------------------
void __fastcall TForm1::N8Click(TObject *Sender)
{
        unsigned short  Attributes;
        unsigned short  NewAttributes;
        AnsiString FileName;
        if (ListView1->SelCount==1){
                FileName=CurrentDir+"\\"+ListView1->Selected->Caption;
                Form2->FileDirName->Caption = FileName ;
                Form2->FilePathName->Caption = CurrentDir+"\\";
                Form2->ChangeDate->Caption = DateTimeToStr
                        (FileDateToDateTime(FileAge(FileName)));
                Attributes = FileGetAttr(FileName);
                Form2->ReadOnly->Checked = Attributes & faReadOnly;
                Form2->Archive->Checked = Attributes & faArchive;
                Form2->System->Checked = Attributes & faSysFile;
                Form2->Hidden->Checked = Attributes & faHidden;
                if (Form2->ShowModal()==IDOK){
                        NewAttributes = Attributes;
                        if (Form2->ReadOnly->Checked)
                                NewAttributes = NewAttributes | faReadOnly;
                        else
                                NewAttributes = NewAttributes & ~faReadOnly;
                        if (Form2->Archive->Checked)
                                NewAttributes = NewAttributes | faArchive;
                        else
                                NewAttributes = NewAttributes & ~faArchive;
                        if (Form2->System->Checked)
                                NewAttributes = NewAttributes | faSysFile;
                        else
                                NewAttributes = NewAttributes & ~faSysFile;
                        if (Form2->Hidden->Checked)
                                NewAttributes = NewAttributes | faHidden;
                        else
                                NewAttributes = NewAttributes  & ~faHidden;
                        if (NewAttributes != Attributes)
                                FileSetAttr(FileName, NewAttributes);
```

```
            }
        }
    }
    //----------------------------------------------------------------------------
```

请注意这里位运算符——“|”、“&”和“~”的使用，通过这几个位运算符实现了文件属性的增加和删除。另外，程序中有这样一行：

```
if (Form2->ShowModal()==IDOK) {……
```

即采取模态方式显示窗体，为了这一行代码能够得以正确执行，编程时必须设定窗体中“确定”按钮的 ModalResult 属性为 mrOK，以保证用户按下“确定”按钮退出后，返回值是 IDOK。

以上已经将“BCB 文件管理器程序”的绝大部分完成，当然，还有一些细节部分的处理请读者参看光盘上的源程序。

虽然上述“BCB 文件管理器”程序已经相当漂亮，但很显然，这个程序与 Windows 95 的资源管理器还是有一定距离的。首先，该程序并没有将每个文件的默认图标显示出来，而只是用自己提供的图标代替；其次，像桌面，网络邻居这些特殊的对象也没有能做出来。然而在一些商业应用程序中，我们确实清楚地看到这些程序的文件浏览部分和 Win32 资源管理器做的一模一样。这是如何做到的？在本章稍后关于 Shell API 部分将讨论这个问题。

3.6　文件流和内存流

本书第 1 章中已经对 C++ Builder 的流类对象进行了概括讨论。实际上，在 C++ Builder 中，从抽象基类 TStream 派生出了各种各样的现实流类，包括 THandleStream、TFileStream、TMemoryStream、TResourceStream、TStringStream 和 TBlobStream 等，这些现实流类分别代表了在各种媒介上或以各种方式存储数据的能力，它们将各种数据类型（包括对象和部件）在内存、外存和数据库字段中的管理操作抽象为对象方法，并且充分利用了面向对象技术的优点，应用程序可以相当容易地在各种 Stream 对象中拷贝数据。

在所有流式对象中，THandleStream、TFileStream 和 TMemoryStream 是最为常用的，这三种流式对象代表了对于内存和外存的数据流处理，以下将重点介绍这三种流式对象。

3.6.1　文件流（TFileStream 类与 THandleStream 类）

C++ Builder 提供两种不同的文件流类：TFileStream 和 THandleStream。TFileStream 类通过文件的文件名来建立文件流，而 THandleStream 类依赖 Windows 文件句柄建立文件流。但实际上，TFileStream 和 THandleStream 在本质上是差不多的。

1. THandleStream 类

THandleStream 类的行为类似于 FileStream 类，但 THandleStream 类定义了 Handle 属性，该属性提供了对文件句柄的只读访问，并且 Handle 属性可以作为 C++ Builder 的 RTL 文件管理

函数和 Windows 文件操作 API 函数的参数，利用这些文件函数来读写数据。THandleStream 的构造函数带有 Handle 参数，该参数指定与 THandleStream 对象相关的文件句柄。THandleStream 类的构造函数声明如下：

```
__fastcall THandleStream(int AHandle);
```

THandleStream 类有以下几个重要的属性和方法：

- Handle 属性。

声明：__property int Handle = {read=FHandle, nodefault};

Handle 属性提供了对文件句柄的只读访问，该句柄由 THandleStream 的构造方法传入。因此除了用 THandleStream 提供的方法外，也可以用文件管理函数和 Windows 文件 API 对句柄进行操作。实际上，THandleStream 类就是运用文件管理函数进行读写操作方法的实现。

- Read、Write 和 Seek 方法。

这三个方法是 TStream 类的虚方法，只是在 THandleStream 中重载了这三个方法，以实现特定媒介——文件的数据存取。

2. TFileStream 类

TFileStream 对象是在磁盘文件上存储数据的现实流式对象。TFileStream 是从 THandleStream 继承下来的，它和 THandleStream 一样都是实现文件的存取操作。不同之处在于 THandleStream 用句柄访问文件，而 TFileStream 用文件名访问文件。实际上 TFileStream 是 THandleStream 上的一层包装，其内核是 THandleStream 的属性和方法。

TFileStream 类中没有增加新的属性和方法。它只是覆盖了构造函数和析构函数。TFileStream 类的构造函数如下：

```
__fastcall TFileStream(const AnsiString FileName, Word Mode);
```

TFileStream 类需要传递一个 Mode 参数，该参数用于设定文件打开时的共享方式和读写权限。这个参数和前面介绍的 FileOpen 文件操作函数的 Mode 参数含义相同，同时可以传递相同的常量值，包括 fmCreate、fmOpenReadWrite、fmShareCompat、fmShareExclusive、fmShareDenyWrite、fmShareDenyRead、fmShareDenyNone、fmOpenRead、fmOpenWrite。

3.6.2 内存流（TMemoryStream 类）

内存流在所有的流式对象中具有特殊的意义。例如，开发人员可以处理内存流，然后将它们保存在一个文件中（内存流类直接提供了 SaveToFile 的方法），或进行相反的操作（LoadFromFile 方法），这种做法可以改进处理大量文件程序的执行速度。

内存流类 TMemoryStream 是一个管理动态内存中的数据的现实流类，它是从 TCustomMemoryStream 类中继承下来的。该类还重载了 Read 和 Write 方法，实现了对内存的读写访问操作，并提供写入、消除内存内容的动态内存管理方法，以及一些用于从磁盘文件的数据交换方法。下面介绍该类的重要属性和方法。

1. TMemoryStream 类的重要属性和方法

- Memory 属性。

声明：__property void * Memory = {read=FMemory};

通过 Memory 属性可以对为内存流分配的内存缓冲区直接访问。

- Clear 方法。

释放内存缓冲区，丢弃和内存流相关的所有数据。调用 Clear 方法等价于同时执行下列三个操作：

设置 Memory 属性为 NULL。

设置 Position 属性为 0。

设置 Size 属性为 0。

- LoadFromFile 方法。

声明：void __fastcall LoadFromFile(const System::AnsiString FileName);

LoadFromFile 方法将 FileName 指定文件的所有内容复制到 MemoryStream 中，并取代已有内容。调用 LoadFromFile 方法后，MemoryStream 将成为文件内容在内存中的完整拷贝。

- LoadFromStream 方法。

声明：void __fastcall LoadFromStream(TStream* Stream);

LoadFromStream 方法将 Stream 指定的流式对象的全部内容复制到 MemoryStream 中，复制过程将取代已有内容。

- SaveToFile 方法。

该方法将内存流中的完整内容写入由 FileName 指定的文件。

- SaveToStream 方法。

该方法将内存流中的完整数据写入 Stream 所指定的流。

2. 内存流和文件流的使用示例

实际上，使用内存流和文件流完全可以替代原有的文件操作方式，并且更为灵活和高效。下面是一个使用流式对象进行文件复制工作的实例，它解释了如何使用流，如图 3-12 所示。

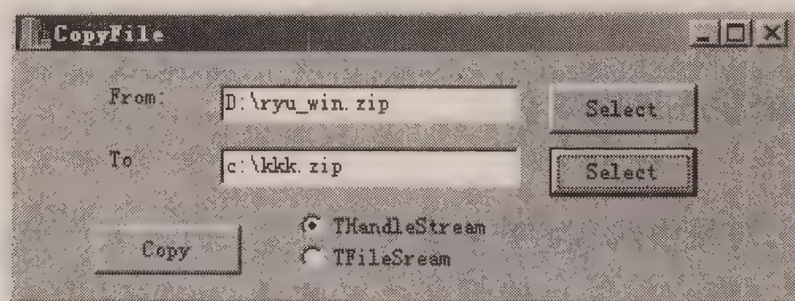


图 3-12　使用内存流和文件流进行复制

Source 使用流式对象复制文件

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
        int iFileHandle;
        if (RadioButton1->Checked){
            iFileHandle = FileOpen(OpenDialog1->FileName, fmOpenRead);
            THandleStream *hStream =
                new THandleStream(iFileHandle);
            TMemoryStream *mStream =
                new TMemoryStream();
            mStream->CopyFrom(hStream,hStream->Size);
            mStream->SaveToFile(Edit2->Text);
            delete mStream;
            delete hStream;
            FileClose(iFileHandle);
            ShowMessage("复制文件成功!");
        }
        else{
            TFileStream *fStream =
                new TFileStream(Edit1->Text,fmOpenRead);
            TMemoryStream *mStream =
                new TMemoryStream();
            mStream->CopyFrom(fStream,fStream->Size);
            mStream->SaveToFile(Edit2->Text);
            delete mStream;
            delete fStream;
            ShowMessage("复制文件成功!");
        }
    }
    //---------------------------------------------------------------------------
```

3.6.3 其他流式对象

流的另外两个重要用法是：直接处理数据库内容的 BLOB（Binary Large Object）大型字段、网络数据的传输。关于使用流类进行网络数据传输在第 11 章和第 12 章有具体的例子，本节着重讨论数据库表格大型字段的处理和 TBlobStream 类。

从数据库开发角度来讲，TBlobStream 类是个很重要的类。TBlobStream 类提供了流式处理 TBlobField、TBytesField 或 TVarBytesField 中数据的技术。编程人员可以象对待文件或流那样在数据库的 BLOB 域中读写数据。

TBlobStream 通过构造函数建立数据库域和 BLOB 流的联接。用 Read 或 Write 方法访问和改变域中的内容；用 Seek 方法，在域中定位；用 Truncate 方法删除域中当前位置起所有的数据。下面介绍 TBlobStream 类几个的重要方法。

1. TBlobStream 的重要方法

- TBlobStream 构造函数。

声明：__fastcall TBlobStream(Db::TBlobField* Field, Db::TBlobStreamMode Mode);

建立 BlobStream 流，并和数据库域联接。其中 Mode 参数说明读写权限，有以下三种取值：

bmRead	BLOB 流允许从数据域中读数据。
bmWrite	BLOB 流允许改变数据域的数据。
bmReadWrite	BLOB 流可以读写数据域数据。

- Truncate 方法。

声明：void __fastcall Truncate(void);

Truncate 方法撤消 TBlobField、TBytesField 或 TVarBytesField 中从当前位置起的数据。

2. TBlobStream 的使用示例

以下一段代码解释了具体如何使用 TBlobStream 类操作数据库数据，它完成了将数据库 1 的 Graphic 字段内容复制到数据库 2 的 Picture 字段。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TBlobStream *Stream1, *Stream2;
    Stream1 = new TBlobStream( (TBlobField*)Table1->FieldByName("Graphic"),
        bmRead);
    try
    {
        Table2->Edit();
        Stream2 = Table2->CreateBlobStream(Table2->FieldByName("Picture"),
            bmReadWrite);
        try {
            Stream2->CopyFrom(Stream1, Stream1->Size);
            Table2->Post();
        }
        __finally {
            delete Stream2;
        }
    }
    __finally {
        delete Stream1;
    }
}
```

3.7　文件相关的高级话题

3.7.1　文件加锁和解锁

在 Windows 操作系统中，文件是共享资源。Windows 中的多个应用程序可以同时打开某一个文件，并从中读取数据或向它写入数据。但应用程序决不能相互覆盖写入，Windows 系统是如何保证文件不被交叉写入的呢？通过文件锁定技术可以解决这个问题。

Win32 API 函数 LockFile 提供了文件锁定的功能。该函数定义如下：

```
BOOL LockFile(
  HANDLE hFile,          // 文件句柄
  DWORD dwFileOffsetLow,        // 锁定区域偏移量低位
  DWORD dwFileOffsetHigh,       // 锁定区域偏移量高位
  DWORD nNumberOfBytesToLockLow,      // 锁定长度低位
  DWORD nNumberOfBytesToLockHigh      // 锁定长度高位
);
```

文件解锁功能由 UnlockFileAPI 函数提供。此函数定义如下：

```
BOOL UnlockFile(
  HANDLE hFile,          // 文件句柄
  DWORD dwFileOffsetLow,        // 锁定区域偏移量低字
  DWORD dwFileOffsetHigh,       // 锁定区域偏移量高字
  DWORD nNumberOfBytesToUnlockLow,  // 锁定长度低字
  DWORD nNumberOfBytesToUnlockHigh  // 锁定长度高字
);
```

下面的范例具有一定实际意义，该范例程序演示加锁和解锁技术的实际应用。该程序可以对硬盘上的文件实施保护，被保护的文件将不能被其他程序以任何形式访问，可以避免重要文件被误删除，以及被他人打开或使用。文件保护程序如图 3-13 所示。

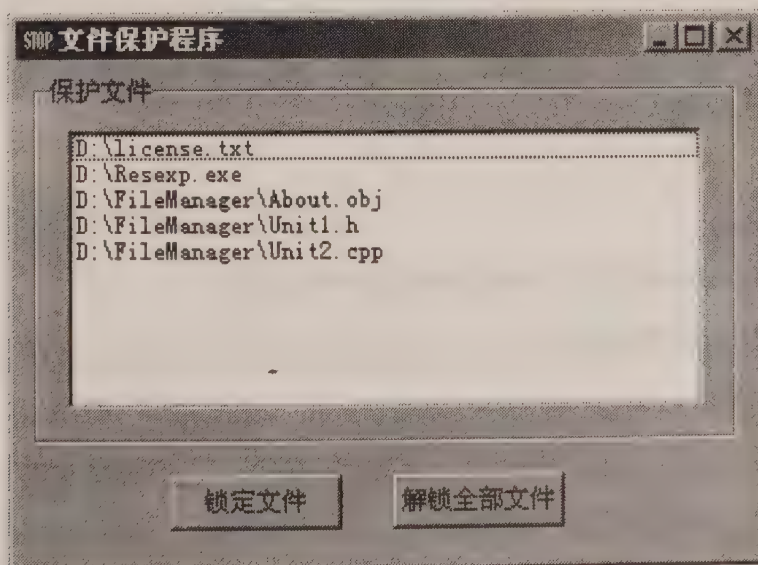


图 3-13　文件保护程序示例

此程序中包含一个主要模块 LockIt，函数的代码如下：

Source 锁定文件操作代码

```
bool __fastcall TForm1::LockIt(char *filename)
{
    HANDLE fHandle;
    try {
        fHandle=CreateFile(filename,
            GENERIC_READ,0,NULL,OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,NULL);
        FileHandle[Index]=fHandle;
        NumLow[Index]=
        GetFileSize(fHandle,&NumHigh[Index]);
        if (NumLow[Index]!=0xffffffff&&NumHigh[Index]!=NULL){
            LockFile(FileHandle[Index],0,0,
            NumLow[Index],NumHigh[Index]);
            Index++;
            return true;
        }
        return false;
    }
    catch (...){
        return false;
    }
}
```

其中使用了 GetFileSize 函数，以获得文件大小的高位字和低位字。然后调用 LockFile 函数锁定文件。

此程序的文件解锁部分如下：

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    for (int i=0;i<Index;i++){
        UnlockFile(FileHandle[i],0,0,NumLow[i],NumHigh[i]);
        CloseHandle(FileHandle[i]);
    }
    Index=0;
    ListBox1->Clear();
}
```

3.7.2 Shell API

随着 Windows 95/98 和 WindowsNT 4.0 的推出，操作系统的用户界面有了整体的提高，这是由于一个完全不同于 Windows 3.X 系列的新的 Windows 外壳（Windows Shell）的出现。

新的 Windows 外壳提供了更加灵活的功能，更加美妙的用户界面风格。然而更为重要的是，新的 Windows 外壳允许应用程序与之紧密结合，从中获取特性并合作完成工作。

前面提到“BCB 文件管理器程序”与 Windows 95 的资源管理器还有很大差距，实际上，应用 Shell API 函数能够完成与 Windows 95 资源管理器完完全全一样的程序。虽然这也是一件颇为麻烦的事，需要用到很多 Shell API 函数和编写冗长的代码，然而，毕竟使用该方法确实可以达到和 Windows 95/98 资源管理器一模一样的效果。

Shell API 函数和大多数 Windows API 函数具有一定的区别。Shell API 函数是基于 COM（Component Object Model，组件对象模型）实现的。关于 COM 技术将在本书的第 16 章具体讲述。

限于篇幅，以下只简单介绍有关 Win32 外壳的具体技术问题。

首先，我们应该了解，Win32 外壳的一个重大变化是桌面的扩展，桌面对象体系不再是只有实际存在的文件和文件夹，还包括了一些特殊的对象，包括我的电脑，网络邻居等等。整个这样一个大的系统，包括桌面上的所有文件，以及实际的文件和文件夹，都被包含在一个外壳名空间中。名空间通过系统与实际对象保持联系。

外壳名空间是以树型结构组织的，在 Windows 95/98 的资源管理器中可以明确的看到这一点，外壳名空间的根节点是“桌面”。而名空间内的所有对象使用指向标志符列表的指针（Pointer to an Identifier List 或 PIDL）标志。

如果想在一个树视图组件内显示所有名空间的对象（像 Windows95 资源管理器一样），必须从根节点即“桌面”开始遍历整个名空间的树结构。为了获得这个开始的起点，应该通过 SHGetDesktopFolder 函数，该函数可以获取一个 IShellFolder 接口，而后可以使用 EnumObjects 函数遍历整个名空间。同时，还可以通过某种方法获得文件的其他信息，包括对象的图标或默认图标等等。


所有 ShellAPI 函数都包含在两个头文件内，分别是 ShellApi.h 和 ShlObj.h。

3.7.3 遍历外壳名空间

根据上小节所述原理，我们这里将具体实现这些想法——即使用 Shell API 遍历并显示 Win32 名空间内的对象，并获得与 Windows 95/98 资源管理器相同的对象信息，以及对象图标或默认图标。并且将以此为基础实现更好的 BCB 文件管理器程序的第 2 版本，见图 3-14。

在 BCB 文件管理器第 2 版中，使用 Shell API 重写了列表视图组件的文件对象显示。首先在程序中定义外壳对象结构 PShellItem，并建立该结构类型的链表 TList*FIDList:

```
struct PShellItem
{
public
        PItemIDList FullID, ID;
        bool Empty;
        AnsiString DisplayName,TypeName,ModDate;
        int ImageIndex,Size;
        DWORD Attributes;
        PShellItem();
}
```

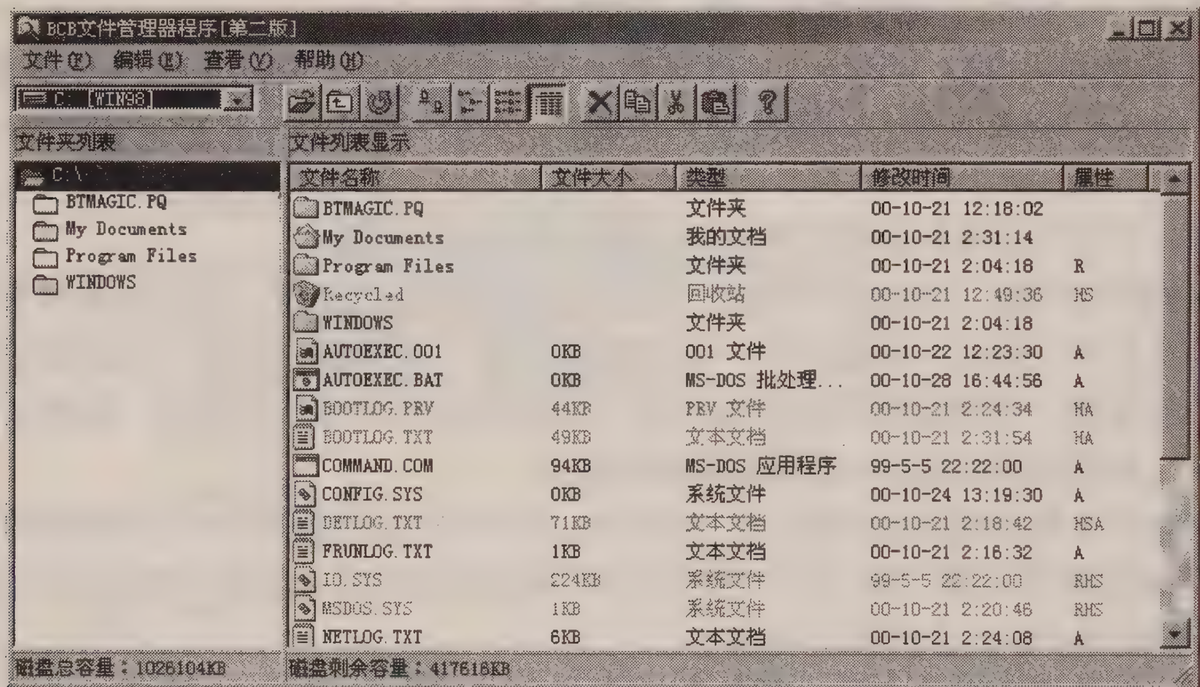


图 3-14　使用 ShellAPI 编写的 BCB 文件管理器的第 2 版

PShellItem 结构的链表对象 FIDList 直接对应列表视图内所应显示的文件对象信息。在本程序中，文件对象信息获得后首先存入 FIDList，然后利用 TListItems 类的 BeginUpdate 和 EndUpdate 方法一次性将信息填入 ListView，从而避免了连续调用 Add 方法造成 ListView 的多次重画。

程序中首先在 FormCreate 函数中调用 SHGetDesktopFolder 函数获取桌面对象的 IShellFolder 接口，并利用 SHGetFileInfo 函数获取外壳空间的图标链表句柄，并通过发送消息的办法将外壳图标链表与列表视图相连。具体程序如下所示：

```
OleCheck(SHGetDesktopFolder(&FIDesktopFolder));
FIShellFolder=FIDesktopFolder;
ImageListHandle=SHGetFileInfo("C:\\ ",0,&FileInfo,sizeof(FileInfo);
        SHGFI_SYSICONINDEX|SHGFI_SMALLICON);
SendMessage(ListView1->Handle,LVM_SETIMAGELIST,LVSIL_SMALL,ImageListHandle);
ImageListHandle=SHGetFileInfo("C:\\ ",0,&FileInfo,sizeof(FileInfo),
        SHGFI_SYSICONINDEX|SHGFI_LARGEICON);
SendMessage(ListView1->Handle,LVM_SETIMAGELIST,LVSIL_NORMAL,ImageListHandle);
```

有了 IShellFolder 接口，我们可以继续实现外壳名空间对象的遍历，程序中当给出一个目录时，将以该目录为基点遍历该目录中的所有对象。

在 UpdateListView 函数中，调用 IShellFolder 接口的 ParseDisplayName 方法获得给出目录名在外壳名空间中的标志符列表指针。然后调用 SetPath 函数，并在 SetPath 函数中获得目标目录的 IShellFolder 接口指针。在 PopulateIDList 函数中调用了 EnumObjects 函数，获得目标目录下所有文件对象的部分信息，并加入到 FIDList 链表。具体实现代码如下：

Source 遍历外壳名空间

```
void__fastcall TMainForm::UpdateListView(AnsiString Value)
{
        wchar_t*P;
        PItemIDList NewPIDL;
        unsigned long int Flags;
        unsigned long int NumChars;
        NumChars=Value.Length();
        Flags=0;
        P=String ToOleStr(Value);
        OleCheck(FIDesktopFolder->ParseDisplayName(Application->Handle,NULL,
                P,&NumChars,&NewPIDL,&Flags));
        SetPath(NewPIDL);
}
//--------------------------------------------------------------------------
void__fastcall TMainForm::SetPath(LPITEMIDLIST ID)
{
        int Index;
        LPSHELLFOLDER NewShellFolder;
        OleCheck(FIDesktopFolder->BindToObject(ID,NULL,IID_IShellFolder,
                (void**)&NewShellFolder));
        ListView1->Items->BeginUpdate();
        try{
                PopulateIDList(NewShellFolder);
                FPIDL=ID;
                FPath=GetDisplayName(FIDesktopFolder,FPIDL,true);
                if(FPath[Fpath.Length()]= ='\\')
                        CurPath=FPath;
                else CurPath=FPath+"\\ ";
                if (ListView1->Items->Count>0)
                {
                        ListView1->Selected=ListView1->Items->Item[0];
```

```
                ListView1->Selected->Focused=true;
                ListView1->Selected->MakeVisible(false);
            }
        }
        __finally
        {
            ListView1->Items->EndUpdate();
        }
}
//---------------------------------------------------------------------------
void__fastcall TMainForm::PopulateIDList(LPSHELLFOLDER ShellFolder)
{
        UINT Flags=
            SHCONTF_FOLDERS|SHCONTF_NONFOLDERS|SHCONTF_INCLUDEHIDDEN;
        PItemIDList ID;
        LPENUMIDLIST EnumList;
        unsigned long NumIDs;
        PShellItem *ShellItem;
        OleCheck(ShellFolder->EnumObjects(Application->Handle,Flags,&EnumList));
        FIShellFolder=ShellFolder;
        ClearIDList();
        while (EnumList->Next(1,&ID,&NumIDs)==S_OK)
        {
            ShellItem=new PShellItem;
            ShellItem->=ID=ID;
            ShellItem->DisplayName=GetDisplayName(FIShellFolder,ID,False);
            ShellItem->Empty=true;
            FIDList->Add (ShellItem);
        }
        FIDList->Sort(ListSortFunc);
        ListView1->Items->Count=FIDList->Count;
        ListView1->Repaint();
}
//---------------------------------------------------------------------------
```

注意到在 SetPath 函数中调出了 ListView 中 ListItems 对象的 BeginUpdate 和 EndUpdate 函数，在这两次调用间 PopulateIDList 函数修改了 ListItems 对象的 Count 属性，并强制 ListView 重画，这将导致 ListView 组件 OnData 和 OnDataHint 事件的发生。OnData 事件发生在某个列表视图项目显示之前，OnDataHint 事件发生在列表视图内容改变时。在这两个事件中，程序才实际向 ListView 组件内写入信息。这样做是为避免使用 Add 方法造成的多次重画，从而极大提高了速度和 ListView 显示的流畅性。请看程序代码：

Source 向列表视图中填入信息

```
void __fastcall TMainForm::ListView1DataHint(TObject *Sender,
    int StartIndex, int EndIndex)
{
    if ((StartIndex < FIDList->Count) && (EndIndex < FIDList->Count))
        CheckShellItems(StartIndex, EndIndex);
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::CheckShellItems(int StartIndex, int EndIndex)
{
    TWin32FindData FileData;
    TSHFileInfo FileInfo;
    TSystemTime SysTime;
    TFileTime LocalFileTime;
    for(int i= StartIndex;i<= EndIndex;i++)
    { // 获得文件对象的其他信息
        if (GetShellItem(i)->Empty){
            GetShellItem(i)->FullID=ConcatPIDLs(FPIDL, GetShellItem(i)->ID);
            GetShellItem(i)->ImageIndex=GetShellImage(GetShellItem(i)->FullID,
                (ListView1->ViewStyle == vsIcon), false);
            SHGetFileInfo(PChar(GetShellItem(i)->FullID), 0, &FileInfo, sizeof(FileInfo),
                SHGFI_TYPENAME | SHGFI_PIDL);
            GetShellItem(i)->TypeName=FileInfo.szTypeName;
            memset(&FileData, 0, sizeof(FileData));
            SHGetDataFromIDList(FIShellFolder, GetShellItem(i)->ID, SHGDFIL_FINDDATA,
                &FileData, sizeof(FileData));
            GetShellItem(i)->Size=FileData.nFileSizeLow/1000;
            memset(&LocalFileTime, 0, sizeof(LocalFileTime));
            if (ValidFileTime(FileData.ftLastWriteTime)&&
            FileTimeToLocalFileTime(&(FileData.ftLastWriteTime),
            &LocalFileTime)&&FileTimeToSystemTime(&LocalFileTime, &SysTime))
            {
            try {
                GetShellItem(i)->ModDate=DateTimeToStr(SystemTimeToDateTime(SysTime));
            }
            catch(EConvertError &E) {
                GetShellItem(i)->ModDate="";
```

```
                }
                }
                else
                        GetShellItem(i)->ModDate="";
                GetShellItem(i)->Attributes=FileData.dwFileAttributes;
                GetShellItem(i)->Empty=false;
            }
        }
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::ListView1Data(TObject *Sender,
      TListItem *Item)
{
        AnsiString Attrs;
        PShellItem * ShellItem;
        if ((Item->Index < FIDList->Count)) {
                ShellItem=GetShellItem(Item->Index);
                Item->Caption=ShellItem->DisplayName;
                Item->ImageIndex=ShellItem->ImageIndex;
                if (ListView1->ViewStyle == vsReport) {
                        if (!(IsFolder(FIShellFolder, ShellItem->ID)))
                                Item->SubItems->Add(Format("%dKB", ARRAYOFCONST((ShellItem->Size))));
                        else
                                Item->SubItems->Add("");
                        Item->SubItems->Add(ShellItem->TypeName);
                        Item->SubItems->Add(ShellItem->ModDate);
                        if (BOOL(ShellItem->Attributes & FILE_ATTRIBUTE_READONLY))
                                Attrs=Attrs + 'R';
                        if (BOOL(ShellItem->Attributes & FILE_ATTRIBUTE_HIDDEN))
                                Attrs=Attrs + 'H';
                        if (BOOL(ShellItem->Attributes & FILE_ATTRIBUTE_SYSTEM))
                                Attrs=Attrs + 'S';
                        if (BOOL(ShellItem->Attributes & FILE_ATTRIBUTE_ARCHIVE))
                                Attrs=Attrs + 'A';
                        Item->SubItems->Add(Attrs);
                }
        }
}
```

这里完成了利用 Shell API 构造的文件浏览列表视图，实际上利用同样的原理实现目录浏览的树视图，唯一困难的地方可能在设置树视图项目图标上。同样利用 SendMessage 函数，发

送 TVM_SETIMAGELIST 消息并传递 TVSIL_NORMAL 参数即可，如下：

```
ImageListHandle=SHGetFileInfo("C:\\", 0, &FileInfo, sizeof(FileInfo),
        SHGFI_SYSICONINDEX | SHGFI_SMALLICON);
SendMessage(TreeView1->Handle, TVM_SETIMAGELIST, TVSIL_NORMAL,ImageListHandle);
```

3.7.4 使用 SHBrowseForFolder 函数和 SHFileOperation 函数

本小节讲述另一方面有关 Shell API 在文件操作和文件浏览的应用，包括用于文件操作的 SHFileOperation 函数、用于浏览文件夹的 SHBrowseForFolder 函数以及如何使用 ShellExecuteEx 函数实现文件属性对话框。

用于浏览文件夹的 SHBrowseForFolder 函数可以用来弹出一个 Win32 风格的标准选择目录的对话框，如图 3-15 所示。

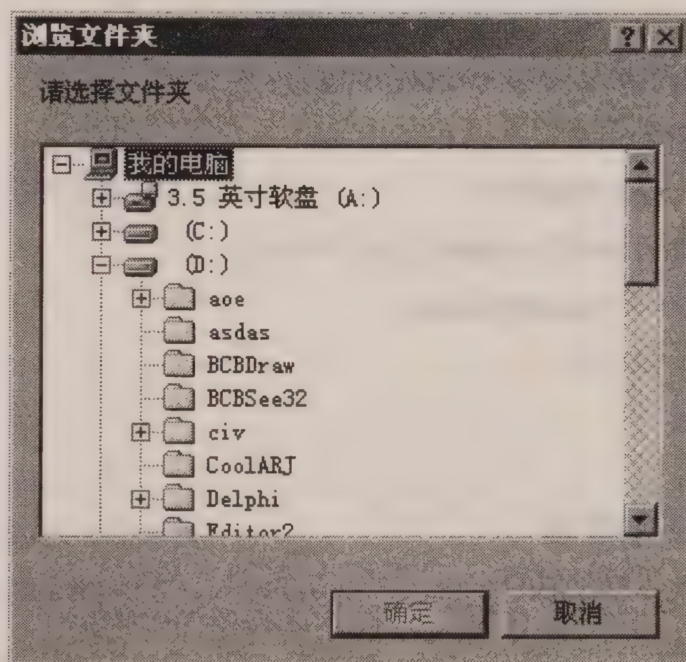


图 3-15　SHBrowseForFolder 函数的应用

然而，使用 SHBrowseForFolder 函数也并不像想象的那样容易，因为为了获得这个函数所需的参数需要做一系列准备工作，包括 SHGetMalloc、SHGetDesktopFolder 等函数的调用。在 BCB 文件管理器第 2 版的程序中，包含一个使用 SHBrowseForFolder 的通用函数，可以不加修改地调用它在任何程序中显示出上图那样的对话框，此函数具体代码如下：

Source　SHBrowseForFolder 函数的调用

```
//---------------------------------------------------------------------------
bool TForm1::BrowseForFolder(AnsiString Title,WideString Root,AnsiString &Directory)
{
    BROWSEINFO BrInfo;
    char *Buffer;
    LPITEMIDLIST RootItemIDList, ItemIDList;
```

```
    LPMALLOC ShellMalloc;
    LPSHELLFOLDER DesktopFolder;
    DWORD Eaten, Flags;
    ZeroMemory(&BrInfo,sizeof(BrInfo));
    if (SHGetMalloc(&ShellMalloc) == S_OK&&ShellMalloc!=NULL){
        Buffer =(char*) ShellMalloc->Alloc(MAX_PATH);
        SHGetDesktopFolder(&DesktopFolder);
        DesktopFolder->ParseDisplayName(Application->Handle,NULL,
            Variant(Root), &Eaten, &RootItemIDList, &Flags);
        BrInfo.hwndOwner = Application->Handle;
        BrInfo.pidlRoot = RootItemIDList;
        BrInfo.pszDisplayName = Buffer;
        BrInfo.lpszTitle = Title.c_str();
        BrInfo.ulFlags = BIF_RETURNONLYFSDIRS;
        ItemIDList = SHBrowseForFolder(&BrInfo);
        if (ItemIDList!=NULL){ // 用户按下"确定"键
            // 把 PIDL 转化为路径名
            SHGetPathFromIDList(ItemIDList, Buffer);
            ShellMalloc->Free(ItemIDList);
            Directory = Buffer;
            return true;
        }
        else { // 用户按下"取消"键
            ShellMalloc->Free(Buffer);
            return false;
        }
    }
}
//-----------------------------------------------------------------------
```

在程序中可以这样调用上面这个函数：BrowseForFolder("浏览文件夹","桌面",Dir)；还可以这样：BrowseForFolder("请选择文件夹","D:\\",Dir); 对话框中的目录树将从不同的节点展开。

BCB 文件管理器第 2 版程序使用 Shell API 函数 SHFileOperation 实现了各种文件操作（见图 3-16）。简单来说，只需把文件操作命令（拷贝、剪切、删除、重命名）发给它，它会像在 Windows 95/98 资源管理器那样完成这个任务。譬如，大文件拷贝时会出现带动画和进度条的对话框显示拷贝状态，拷贝重名时会有对话框提示，删除文件时它并不直接删除而是提示将文件放入回收站等等。

这个函数原型如下：

```
WINSHELLAPI int WINAPI SHFileOperation(
  LPSHFILEOPSTRUCT lpFileOp
);
```

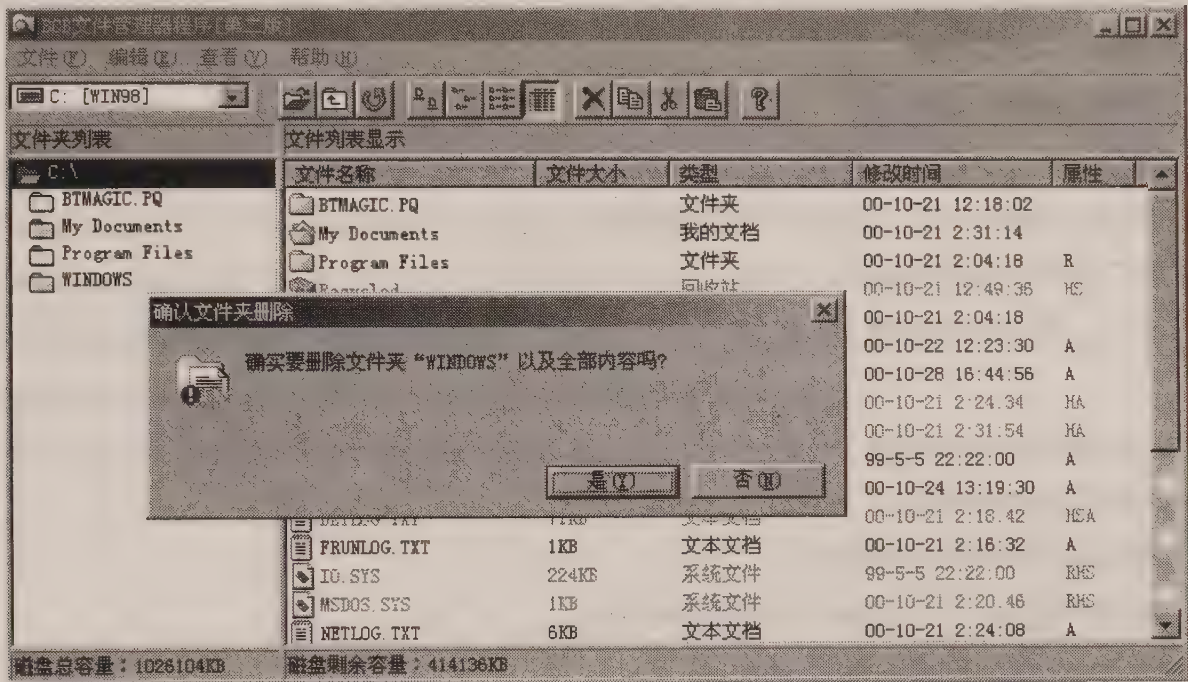


图 3-16 SHFileOperation 函数的应用

函数仅有一个参数，此参数包含执行文件操作命令的所有信息。参数类型的结构是这样定义的：

```
typedef struct _SHFILEOPSTRUCT
  HWND       hwnd;
  UINT       wFunc;
  LPCSTR     pFrom;
  LPCSTR     pTo;
  FILEOP_FLAGS fFlags;
  BOOL       fAnyOperationsAborted;
  LPVOID     hNameMappings;
  LPCSTR     lpszProgressTitle;
} SHFILEOPSTRUCT;
```

其中 hwnd 参数是显示对话框的句柄。wFunc 指明操作的类型。支持四种操作：FO_COPY 拷贝、FO_MOVE 剪切、FO_DELETE 删除、FO_RENAME 重命名。pFrom 参数和 pTo 参数，分别代表原文件全名（包括路径）和目标路径（或操作后文件全名），在 FO_DELETE 操作中 pTo 参数不起作用。fFlags 参数是风格选项，可取值很多，在此不再一一介绍。

对于这个演示程序来说，其中的文件列表框有一个 PopMemu，可以进行拷贝、剪切、粘贴和删除四种操作，现将粘贴和删除的函数列出，请注意 SHFileOperation 函数的使用。

Source SHFileOperation 函数的调用

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
      TSHFileOpstructfileOp;
```

```
        ZeroMemory(8,file Op),Sizeof(TSHFileOpStruct);
        if (ComState==csCopy){
            fileOp.hwnd=Handle;
            fileOp.wFunc=FO_COPY;
            fileOp.fFlags=FOF_SIMPLEPROGRESS;
            fileOp.pFrom=FileName.c_str();
            fileOp.pTo=CurrentDir.c_str();
        }
        else if (ComState==csCut){
            fileOp.hwnd=Handle;
            fileOp.wFunc=FO_MOVE;
            fileOp.fFlags=FOF_SIMPLEPROGRESS;
            fileOp.pFrom=FileName.c_str();
            fileOp.pTo=CurrentDir.c_str();
        }
        if (ComState!=csNone){
            ComState=csNone;
            SHFileOperation(&fileOp);
            UpdateListView(CurPath);
        }
    }
    //---------------------------------------------------------------------------
    void __fastcall TMainForm::D1Click(TObject *Sender)
    {
        TSHFileOpStruct fileOp;
        ZeroMemory(&fileOp,sizeof(TSHFileOpStruct));
        if (ListView1->Selected!=NULL) {
            AnsiString fName=CurPath+GetShellItem(ListView1->Selected->Index)->DisplayName;
            fileOp.hwnd=Handle;
            fileOp.wFunc=FO_DELETE;
            fileOp.fFlags=FOF_SIMPLEPROGRESS;
            fileOp.pFrom=fName.c_str();
            SHFileOperation(&fileOp);
            UpdateListView(CurPath);
        }
    }
```

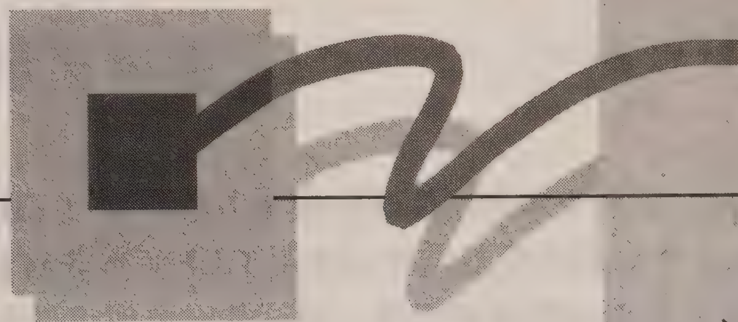
最后说明如何使用 ShellExecuteEx 函数实现 Windows 默认的文件属性对话框。ShellExecuteEx 函数与前面用到的 ShellExecute 函数差不多，但 ShellExecuteEx 函数只需传递一个 SHELLEXECUTEINFO 结构的指针作为参数，该结构的 lpVerb 属性指定 ShellExecuteEx 执行的动作，如果需要显示文件属性对话框则需赋 lpVerb 为 properties，具体代码如下：

```
void __fastcall TMainForm::R1Click(TObject *Sender)
{
    TShellExecuteInfo ShellExecuteInfo;
    ZeroMemory(&ShellExecuteInfo,sizeof(TShellExecuteInfo));
    ShellExecuteInfo.cbSize=sizeof(TShellExecuteInfo);
    ShellExecuteInfo.lpFile=(CurPath+ListView1->Selected->Caption).c_str();
    ShellExecuteInfo.lpVerb="properties";
    ShellExecuteInfo.fMask=SEE_MASK_INVOKEIDLIST;
    ShellExecuteEx(&ShellExecuteInfo);
}
```

第4章

可与 ACDSee 媲美的 BCBSee32
——深入图像文件编程

Windows 图像编程技术
TImage 组件和 TPicture 对象
TBitmap、TIcon 和 TMetafile 对象
处理 JPEG 格式图像
图像浏览
图像格式转换和图像打印



本章导读

图形图像编程一直都是一个非常有趣的话题，在 Windows 时代更是如此。Windows 编程中经常需要更多的图像使得应用程序更加友好。另一方面，Windows 系统也使我们可以更加容易使用图像。

本章通过对一个精彩范例——BCBSee32 图像浏览和转换工具的分析，讲述 C++ Builder 多格式图像显示、图像特性控制、格式转换等编程技术。BCBSee32 以著名的 ACDSee 图像浏览工具为蓝本，支持 Bitmap、JPEG、Metafile、Icon 等多种图像格式，实现丰富的图像浏览功能、图像转换功能和相关的辅助功能（如设为墙纸，打印图像等）。相比之下，BCBSee32 比 ACDSee 工具毫不逊色。

本章将进入激动人心的图形图像编程世界，并展示 C++ Builder 强大的图形图像编程功能，内容包括:

- Windows 图像编程的一般讨论以及 TImage 组件和 TPicture 对象。
- TBitmap、TIcon、TMetafile 和 TJPEGImage 对象。
- 实现图像浏览和转换工具 BCBSee32。通过此范例程序的实现来阐述图像浏览、转换以及图像打印的编程技术。

4.1 图像显示技术

Windows 应用程序使用图形的根本方法是通过 GDI（图形设备接口），而在 C++ Builder 中，TCanvas 对象的出色封装使得使用 GDI 变得非常容易。同时 C++ Builder 还提供了一系列用于处理和显示图形图像的组件，而这些对象或组件可称得上是整个 VCL 可视化组件库中最为精彩的部分之一。

在图形图像编程中，一个首要的问题是如何将图像数据在屏幕上直观的表现出来，即图像显示。应用 Borland C++ Builder 进行图像编程在这一方面有明显的优势，TImage 组件可以容易的显示多种格式图像，而 TBitmap、TIcon、TMetaFile 和 TJPEGImage 类封装了图像文件格式的细节，可以使我们直接应用这些格式的文件，并专注于图形图像功能的开发。

在进一步认识 C++ Builder 图形类之前，了解 Windows 图形编程机制是有必要的，这将使你从另一高度掌握 C++ Builder 的图形类。

4.1.1 Windows 图形设备接口

Windows 是一个基于图形用户界面（GUI）的操作系统，这使得 Windows 应用程序有漂亮而又统一的用户界面。对于一个程序员来说，由于 Win32 中图形显示实现了设备无关性，为 Win32 编写的程序可以在多种设备上绘制和输出，因此完全不用理会恼人的硬件设备特性。

Windows 使用图形设备接口（GUI）以及设备驱动程序来实现设备无关的图形。应用程序通过创建设备描述表与 GUI 交互。设备描述表有以下四类：

- 显示设备描述表，用于支持在显示器上的绘制操作。
- 打印设备描述表，用于支持在打印机或绘图仪上的绘制操作。
- 内存设备描述表，用于支持在内存中进行绘制操作。
- 信息设备描述表，用于支持设备数据的访问。

一般来说，Windows API 的绘图编程过程如下：

```
HDC hDC;
PAINTSTRUCT ps;
// 获取设备描述表句柄
hDC=GetDC(hwnd);
BeginPaint(hwnd,&ps);
......
EndPaint(hwnd,&ps);
```

其中在 BeginPaint 和 EndPaint 函数之间的绘图函数要使用 hDC 设备描述表句柄。而在 C++ Builder 中编程人员完全不必考虑这些问题，C++ Builder 封装的 TCanvas 对象将自动管理

设备描述表句柄。如果仍然不得不使用 Windows API 绘图函数时，只需访问 TCanvas 对象的 Handle 属性，它正是你想要的设备描述表句柄。

有关 TCanvas 的问题将放到下一章具体讨论。本章我们将关注 C++ Builder 中的 TImage 组件及其相关问题。

4.1.2 TImage 组件

TImage 组件是 C++ Builder 中使用频率极高的组件，它的主要功能是图像显示。通过 TImage 组件，可以容易的装入多种格式的图像文件（包括 JPEG 图像文件、位图、图标以及图元文件和增强图元文件），显示图像并实现简单的处理功能。TImage 组件如图 4-1 所示。

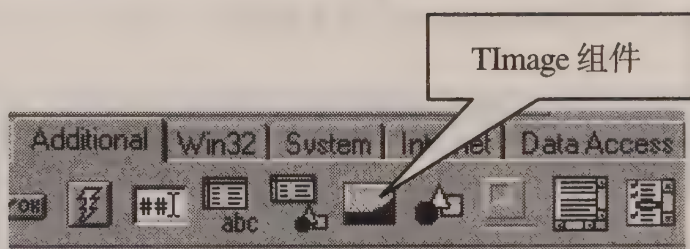


图 4-1 Additional 页上的 TImage 组件

TImage 组件的主要属性如下：

- Canvas 属性。

声明：__property Graphics::TCanvas* Canvas = {read=GetCanvas};

图像画布。为 TCanvas 类型，包含目前所显示的信息。关于 TCanvas 类在下一章中还将详细介绍。

- Picture 属性。

声明：__property Graphics::TPicture* Picture = {read=FPicture, write =SetPicture};

保存在 TImage 组件显示的图像（数据）。请注意它与 Canvas 属性的不同。在设计期间，可在 Object Inspector 中通过 Picture 属性将图像文件装入。

- Stretch 属性。

决定图像显示时是否进行拉伸。当 Stretch 为真时，位图像将根据组件的大小调整自身的大小，当组件大小改变时，图像文件也做相应变化。但 Stretch 属性对图标没有作用。

- AutoSize 属性。

决定 TImage 组件是否与图像大小保持一致。当 AutoSize 为假值时，不论图像有多大，组件将保持设计时的大小。如果组件比图像小， 那么只有一部分图像是可见的。注意当 AutoSize 为真值时，Stretch 属性将不起作用（即拉伸无法进行）。

- Height/Width 属性。

这两个属性继承自 TControl 类，但它们对于 TImage 组件至关重要。Stretch 属性需要与 Height/Width 属性配合使用。

- Transparent 属性。

是否透明显示。对于具备 Mask 的位图、元图和图标起作用。

TImage 组件的关键属性是 Picture 属性。TPicture 类现实图形对象的封装，也就是说，TPicture 类中包含的可以是一个位图、元图文件、图标、JPEG 格式图像文件或者是用户自定义的图形类型。通过 TPicture 类可以使用相同的方法来实现对于各种格式图像的操作。

TPicture 类提供了一整套图像文件的存取方法。包括：LoadFromClipboardFormat、LoadFromFile、RegisterClipboardFormat、RegisterFileFormat、RegisterFileFormatRes、SaveToClipboardFormat、SaveToFile、SupportsClipboardFormat，其中最为常用的是 LoadFromClipboardFormat、SaveToClipboardFormat 和 LoadFromFile、SaveToFile，支持图像从文件或剪贴板读入。

要保存一个位图，可以用 SaveToFile 方法；要把图像复制到剪切板，可以调用 TClipboard 对象的 Assign 方法。

下面提供一个简单的程序以演示 TImage 组件的应用。这个程序的作用是读入一个图像文件，并实现图像的放大和缩小。该示例程序如图 4-2 所示。

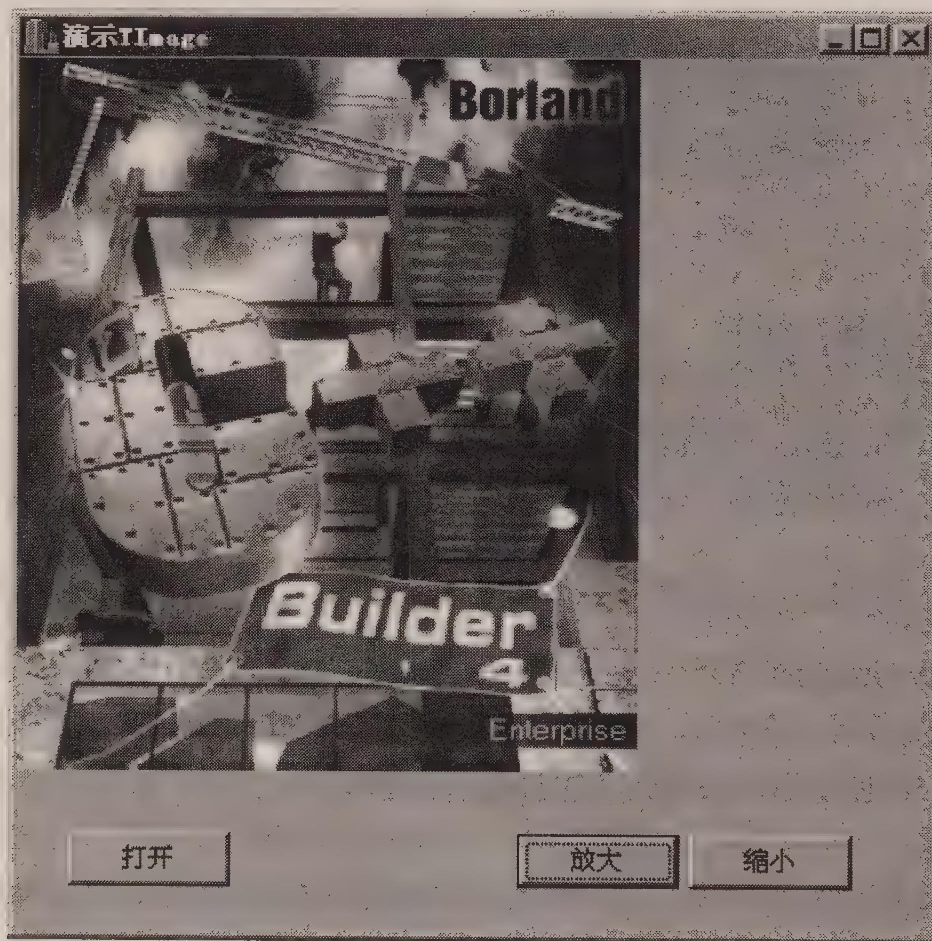


图 4-2　TImage 组件示例程序

使用 TImage 组件的 Picture 属性的 LoadFromFile 方法将图像文件导入。

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    OpenDialog1->Filter = GraphicFilter(__classid(TGraphic));
    if (OpenDialog1->Execute())
        Image1->Picture->LoadFromFile(OpenDialog1->FileName);
}
```

这里需要特别说明的是，如果想支持打开 JPEG 图像文件，必须手工在程序头文件中加入：

```
#include <jpeg.hpp>
```

实现图像缩放的原理很简单，即在锁定 Stretch 属性为 true 时，改变 Image 组件的 Width 和 Height 属性。例如实现图像放大的代码为：

```
Image1->Width*=1.1;
Image1->Height*=1.1;
```

4.2 现实图形对象

TPicture 对象的实现基于一系列现实的图形对象，包括 TBitmap，TIcon，TMetafile，TJPEGImage 或用户自定义图像。TPicture 对象包含了对应于 Tbitmap、Ticon、TMetafile 对象的属性 Bitmap、Icon、Metafile，但同一时刻一个 TPicture 对象只能含有某一类的现实图形对象。

4.2.1 TGraphic 类

TGraphic 是一切现实图形对象的基类，TPicture 对象包含了 TGraphic 类型的属性 Graphic，通过它可以获得当前图像的一些特性，并明确当前装入的图像到底是 Icon、Bitmap、Metafile 还是用户自定义图像。

TPicture 对象中真正起主导作用的属性就是 Graphic，它包含了图形文件数据。所以直接对 Graphic 属性进行赋值就可以装入一个图像。例如：

```
Graphics::TBitmap  *bitmap = new Graphics::Tbitmap();
bitmap->LoadFromFile("D:\\hahaha.bmp");
Picture1->Graphic = bitmap;
```

完全等价于：

```
Picture1->LoadFromFile("D:\\hahaha.bmp");
```

或：

```
Picture1->Bitmap->LoadFromFile("D:\\hahaha.bmp");
```

Graphic 属性在图形绘制中是常用到的，例如将一个 Image 组件中的图像画到一个 PaintBox 中，可以：

```
PaintBox1->Canvas->Draw(0,0,Image1->Picture1->Graphic);
```

TGraphic 类的重要属性包括调色板 Palette 和调整标志 Modified。

- Palette 属性。

声明：__property HPALETTE Palette = {read=GetPalette, write=SetPalette, nodefault};

可以返回或设定图像的调色板。类型为 HPALETTE，但图像不需要调色板或没有使用调色板时返回 0 值。

- Modified 属性。

声明：__property bool Modified = {read=FModified, write=SetModified, nodefault};

当图像有所改变时，此属性为真值。如果其值为 false，则表明图像从装入以来没有变化。注意此属性对于 TGraphic 类中包含的是一个 Bitmap 图形时才正确。

4.2.2 TBitmap 类

位图（Bitmap）在图形图像编程中有举足轻重的地位。位图（Bitmap）是一个强有力的图像对象，常用于在内存或磁盘上建立和存储图像，并设计为能够快速装入、处理和显示。

TBitmap 类是 Windows 位图（HBITMAP） 及其相关处理的一个封装。但不仅仅如此，TBitmap 类还封装了 Windows 调色板（HPALETTE）的相关内容，TBitmap 可以自动管理和实现调色板。

下面介绍 TBitmap 类的重要属性。

- Canvas 属性。

声明：__property TCanvas* Canvas = {read=GetCanvas};

位图画布。这是 TBitmap 类与众不同的一个属性，其他现实图像对象都不具备此属性。Canvas 属性允许将位图看作一个假想的画布对象，这就意味着可以使用 TCanvas 类提供的属性和方法对位图对象进行操作，如访问像素，使用绘图函数，以及使用 Draw 和 StretchDraw 等等（关于 TCanvas 类的讲述请参看第 5 章），这一切给编程人员操纵位图带来了极大的灵活性。

- Handle 属性。

声明：__property HBITMAP Handle = {read=GetHandle, write=SetHandle, nodefault};

位图句柄。它提供给编程人员进入 Windows GDI 位图对象的句柄。这是在不得不使用 Windows API 进行绘图时所必须访问的属性。

- HandleType 属性。

句柄类型。决定位图是一个 DDB（Device Dependent Bitmap，设备相关位图）还是一个 DIB（Device Independent Bitmap，设备无关位图）。

- IgnorePalette 属性。

决定 TBitmap 类是否自动管理和实现当前位图的调色板。如果此值为 true，TBitmap 类将不再自动实现调色板而使用系统默认调色板。

- MaskHandle 属性。

位图的掩图句柄。

- Monochrome 属性。

布尔值。决定是否以黑白模式显示位图。

- Palette 属性。

可以返回或设定图像的调色板。类型为 HPALETTE，但图像无调色板时为 0 值。

- PixelFormat 属性。

位图对象的颜色深度，为 TPixelFormat 枚举类型。其值可以是：pfDevice、pf1bit、pf4bit、pf8bit、pf15bit、pf16bit、pf24bit、pf32bit 和 pfCustom。访问 PixelFormat 可以获取或设置当前位图的颜色深度，这也会改变位图的存储模式，并对 ScanLine 属性的使用有影响。

- ScanLine 属性。

声明：__property void * ScanLine[int Row] = {read=GetScanline};

用于访问位图中一整行的像素。利用 ScanLine 属性可以使图像处理速度大大提升，关于这一点将在第 5 章中具体讨论。注意这个属性只能对 DIB 设备无关位图使用。

- TransparentColor 属性。

设定用作透明的颜色（TransparentColor 所指定的颜色在显示时将不被绘制）。

- TransparentMode 属性。

声明：__property TransparentMode = {read=FTransparentMode, write=SetTransparentMode, default=0};

设定透明模式。可选值为 tmAuto 和 tmFixed。当 TransparentMode 属性值为 tmAuto 时，TransparentColor 将被指定为位图的左下角的像素颜色。

TBitmap 类提供了 LoadFromClipboardFormat、SaveToClipboardFormat 和 LoadFromFile、SaveToFile 方法，支持图像和文件或剪贴板之间读入和存储。另外 TBitmap 还提供流式存取方法，这是 TPicture 类型所不具备的。

TBitmap 类的使用非常灵活，在图像处理方面有广泛的应用，有很多可说的地方。你将能在本章以及第 5 章、第 6 章的实例编程中体会到这一点。

4.2.3 TIcon 类和 TMetafile 类

TIcon 类和 TMetafile 类分别用于另两类现实图形对象——Icon 图标和 Metafile 图元文件。

1. TIcon 类

TIcon 类封装了 Windows 图标对象。Icon 图标文件的扩展名一般为*.ico，图标其实是带有掩图的特殊位图。注意 Icon 是不能被 Stretch 的——在 TImage 中的 Stretch 对于 Image 中的现实图形为 Icon 时是无效的。同时也只能使用 Draw 方法把一个图标画在一个画布对象上，而不能使用 StretchDraw 方法。

以下介绍 TIcon 类的重要属性。

- Handle 属性。

声明：__property HICON Handle = {read=GetHandle, write=SetHandle};

图标句柄。它提供进入 Windows GDI 图标的句柄，在使用 Windows API 函数处理图标时，必须使用此属性。

- Transparent 属性。

表明是否进行透明显示。此属性实际上是只读的。

- Palette 属性。

可以返回或设定图像的调色板。类型为 HPALETTE，图像无调色板时为 0 值。

- PaletteModified 属性。

表明调色板是否被改动。

TIcon 类同样提供了 LoadFromClipboardFormat、SaveToClipboardFormat、LoadFromFile 和 SaveToFile 等方法。

2. TMetafile 类

TMetafile 类用于处理 Windows 3.X 的图元文件（Metafile）和 Win32 的增强图元文件

（Win32 Enhanced Metafile），这两种图像文件的扩展名分别为.WMF 和.EMF。

Mircosoft Office 中的剪贴画就是以图元文件格式存储的。图元文件可以包含一些特殊的信息，包括图像作者、文件描述等等，在 C++ Builder 的 TMetafile 类中分别对应了 CreatedBy 和 Description 属性。

TMetafile 类的一个重要属性是 Enhanced，如果此属性为 true，则图元作为增强图元处理，否则作为 Windows 3.X 图元处理。增强图元格式是图元在 Win32 中的扩展版本，它存储了与映射模式以及系统有关的信息。

与 TBitmap 类和 TIcon 类相同，TMetafile 类也提供了 LoadFromClipboardFormat、SaveToClipboardFormat 和 LoadFromFile、SaveToFile 等存取方法。

虽然 Metafile 格式的图像并不十分常用，但它的意义十分重大。Metafile 格式图像是唯一一种被 Windows 系统支持的矢量图形格式。矢量图形格式和一般的像素点阵图像格式是完全不同的，矢量图形格式抽象的存储重建图形所需的信息，包括每条线的起始点和终止点，或定义一条曲线的数学表示。矢量图像格式可以非常小，并且可以被任意放缩。

Windows 图元文件基本上就是对 Windows GDI 函数的一系列调用。在存储了这一系列调用之后，可以重新使用它们重建图像。C++ Builder 通过 TMetafile 类和另一个与图元格式图像密切相关的类——TMetafileCanvas 类支持 Windows 图元文件。TMetafileCanvas 类如同一个 TCanvas，实际上，TMetafileCanvas 就是继承于 TCanvas 类的，可以像使用 TCanvas 类那样在 TMetafileCanvas 类的实例上作图，包括使用各种画点、画线、画各种几何图形的方法，使用与 MetafileCanvas 相关的 TMetafile 对象的 SaveToFile 方法，就可以将所作的图形以矢量图形的方式保存。

下面是一个使用 TMetafile 类的实例程序，该程序在 MetafileCanvas 上绘制了一个图形，并保存为一个 Windows 图元文件，同时通过 TPaintBox 组件将绘制的图形显示出来。程序的具体代码如下：

Source 使用 Windows 图元格式

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Wmf = new TMetafile;
    WmfCanvas = new TMetafileCanvas(Wmf, 0);
    Rotation = 0;
    PointCount = MaxPoints;
    WmfCanvas->Pen->Color = clTeal;
    WmfCanvas->Brush->Style=bsClear;
    const float M_2PI = 2 * M_PI;
    float StepAngle = M_2PI / PointCount;
    Rotation += M_2PI / 32;
    if (Rotation > StepAngle)
        Rotation -= StepAngle;
    float j;
```

```
        int i=0;
        for (i = 0,j=Rotation; i < PointCount; i++, j += StepAngle)
        {
            Points[i].X = cos(j);
            Points[i].Y = sin(j);
        }
        WmfCanvas->Ellipse(0, 0, 300, 300);
        for (int i = 0; i < PointCount; i++) {
            for (int j = i + 1; j < PointCount; j++) {
                WmfCanvas->MoveTo(150 + floor(Points[i].X *150),
                    150 + floor(Points[i].Y *150));
                WmfCanvas->LineTo(150 + floor(Points[j].X *150),
                    150 + floor(Points[j].Y *150));
            }
        }
        delete WmfCanvas;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        PaintBox1->Canvas->Draw(0,0,Wmf);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        Wmf->SaveToFile(ExtractFilePath(Application->ExeName)+"demo.wmf");
}
//---------------------------------------------------------------------------
```

该范例程序的执行结果如图 4-3 所示。

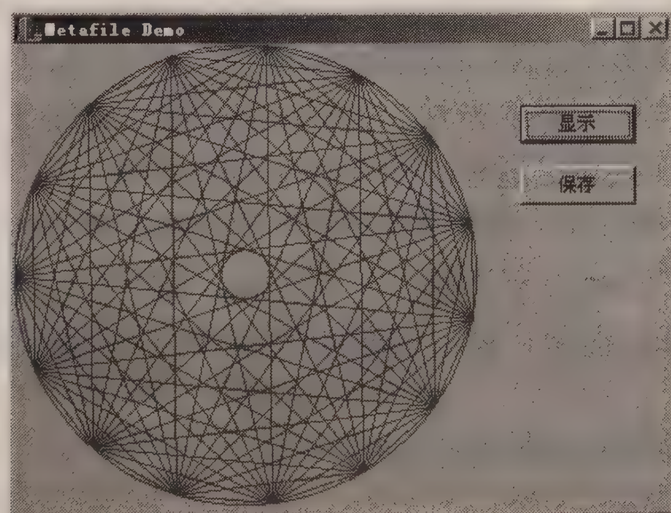


图 4-3　使用 TMetafile 和 TMetafileCanvas 操作图元文件

矢量图像格式具有非同寻常的意义。例如，如果我们要开发类似 AutoCAD 这样的应用程序，就可以使用 Windows 图元作为存储格式。一个矢量图像格式是可以被任意放缩的，当使用 StretchDraw 绘制一个图元图像时，并不会使画质降低，这与其他图像格式并不相同。这是因为图元文件是通过改变坐标映射来改变显示比例的。

4.3 使用和控制 JPEG 格式图像

在 C++ Builder 4 中可以直接使用 JPEG 格式（Joint Photographic Experts Group，联合摄影专家组）图像。JPEG 采用了一种有损压缩的算法，使得图像容量大大减小。如果在程序中使用位图，一个稍大一些的真彩图像，容量往往就在 1MB 以上，这将使程序变得非常臃肿；而如果使用 JPEG 图像，较大的真彩图像往往才几十千字节，在空间方面是相当合算的。

4.3.1 功能强劲的 TJPEGImage 类

C++ Builder 通过 TJPEGImage 类提供对 JPEG 格式图像的支持。TJPEGImage 包含在“jpeg.hpp”头文件中。

TJPEGImage 是一个非常有用并且有点特殊的类，同时它也是为数不多的在 C++ Builder 联机帮助中没有提到的类。所以，本章将 TJPEGImage 类单独讲述。

TJPEGImage 类继承于 TGraphic 类，TJPEGImage 类可以用来读或写 JPEG 压缩图像数据。TJPEGImage 处理静止数字图像的压缩和还原，它从一个 TJPEGData 类的实例为媒介获得 JPEG 图像的实际数据。每一个 JPEG 图像对象可以通过它的 TJPEGData 对象和其他 JPEG 图像对象实例共享数据，这是通过 TJPEGImage 类的 Assign 方法生成一个使用副本来实现的。

TJPEGImage 对象有一个内在的描述 JPEG 图像的位图（Bitmap）对象。这个内在图像和 JPEG 的原始数据都是只读的。TJPEGImage 类包含了一些用来处理颜色转换、压缩、还原以及图像性能的属性。

一个 TJPEGImage 对象有如下特点：

- 不拥有画布属性。但是，TJPEGImage 可以执行 Draw 和 StrecthDraw 方法将其自身作为一个 TGraphic 对象画在其他对象的画布上。
- 不提供进入其内在位图图像的方法。内在位图仅仅为 JPEG 图像而创建。
- 执行涉及计算和处理共享是通过一个 TJPEGData 对象。

可以使用 TPicture 的 LoadFromFile 方法来装入一个 JPEG 类型的图像。但与其他现实图形对象（如 TBitmap，TIcon，TMetafile）不同，在 TPicture 类中并没有提供一个属性直接访问包含在一个 Picture 对象中的 JPEG 图像，此时控制和处理 JPEG 图像只能通过 TPicture 的 Graphic 属性。

以下讨论有关 TJPEGImage 的主要属性和方法。

- CompressionQuality 属性。

声明：typedef Shortint TJPEGQualityRange;

__property TJPEGQualityRange CompressionQuality = {read=FQuality, write=FQuality, nodefault};

描述 JPEG 图像的压缩质量。其值在 1 到 100 之间。数值越高文件压缩质量越好，压缩率越低，而文件尺寸也越大。

- Grayscale 属性。

决定是否使用灰阶（Grayscale）来显示 JPEG 图像，为布尔值。设置此属性为 true 可以加快显示速度，但输出图像值包括 255 级灰度。

- Palette 属性。

声明：__property HPALETTE Palette = {read=GetPalette, write=SetPalette, nodefault};

可以返回或设定 JPEG 图像的调色板，类型为 HPALETTE。

- Performance 属性。

声明：enum TJPEGPerformance { jpBestQuality, jpBestSpeed };

__property TJPEGPerformance Performance = {read=FPerformance, write=SetPerformance, nodefault};

描述 JPEG 图像的特性，决定是速度优先还是质量优先。

- PixelFormat 属性。

声明：enum TJPEGPixelFormat { jf24Bit, jf8Bit };

__property TJPEGPixelFormat PixelFormat = {read=FPixelFormat, write=SetPixelFormat, nodefault};

描述 JPEG 图像像素点的颜色深度，包括 8 位和 24 位两种方式。

- ProgressiveDisplay 属性。

决定还原 JPEG 图像所用的显示方式。如果此属性值为 true，图像将随着数据的不断读入而一点一点的显示出来，但会减慢显示速度。在浏览器中浏览网页时，网页中 JPEG 图像就采用了 ProgressiveDisplay 的显示方式。

- Scale 属性。

声明：enum TJPEGScale {jsFullSize, jsHalf, jsQuarter, jsEighth };

__property TJPEGScale Scale = {read=FScale, write=SetScale, nodefault};

决定 JPEG 图像以何种倍数尺寸显示。包括原尺寸、原尺寸的一半、原尺寸的 1/4、原尺寸的 1/8 四种模式。

- Smoothing 属性。

决定 JPEG 图像是否以光滑方式显示。

TJPEGImage 类除了提供 LoadFromFile 和 SaveFromFile 读写方法外，还包括了几个特殊的方法：

- Assign 方法。

声明：virtual void __fastcall Assign (Classes::TPersistent* Source);

复制一个 JPEG 图像对象，并建立一个 JPEG 内部数据对象的关联，此方法为 TPersistent 抽象类的 Assign 方法的重载。

- Compress 方法。

进行 JPEG 图像的强制压缩。

4.3.2 TJPEGImage 应用示例

C++ Builder 5 中并未为 TJPEGImage 类提供帮助，所以直接阅读头文件可能是充分了解 TJPEGImage 类的最佳方式。关于 TJPEGImage 类的定义及实现源码位于 C++ Builder 目录下的 Include\vcl\jpeg.hpp 文件中，阅读此文件将更清晰地了解 TJPEGImage 类的结构与实现。

C++ Builder 提供的 TJPEGImage 类是一个相当强大的类，使用它可以实现有关 JPEG 图像的所有控制和操作。这正是 C++ Builder 提供 JPEG 支持和 Visual Basic 提供 JPEG 支持的不同。

下面将通过一个范例程序演示如何利用 TJPEGImage 类控制 JPEG 图像特性。而在本章的主要范例程序 BCBSee32 中，还将包括使用 TJPEGImage 类将各种格式的图像转换为 JPEG 图像的技术。

这个范例是 TJPEGImage 各个控制属性的直接应用，其运行结果如图 4-4 所示。

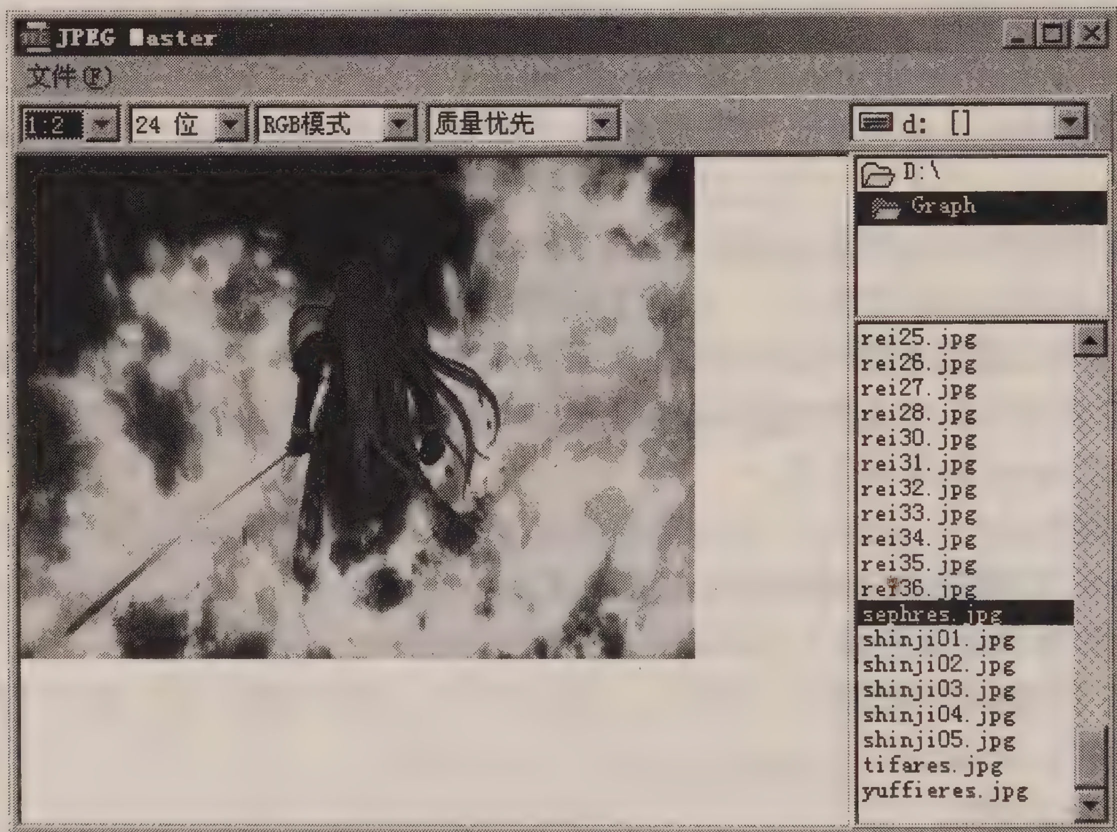


图 4-4 JPEG Master 示例程序

程序中，首先是进行必要的初始化工作。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Scale->ItemIndex = 0;
    PixelFormat->ItemIndex = 0;
    ColorSpace->ItemIndex = 0;
    Performance->ItemIndex = 0;
    OpenDialog1->Filter = GraphicFilter(__classid(TGraphic));
```

```
        FileListBox1->Mask = GraphicFileMask(__classid(TGraphic));
}
```

当用户单击右边的文件列表框时，在 Image1 组件中装入图像。

```
void __fastcall TForm1::FileListBox1Click(TObject *Sender)
{
        Image1->Picture->LoadFromFile(FileListBox1->FileName);
        SetJPEGOptions(this);
}
```

SetJPEGOptions 是本示例程序中的关键函数，它用于实现对 JPEG 图像的控制。它同时是上方四个 ComboBox 组合框的 OnClick 事件的响应函数。以下是 SetJPEGOptions 的实现：

Source 使用 TJPEGImage 类显示和控制 JPEG 格式图像

```
//---------------------------------------------------------------------------

void __fastcall TForm1::SetJPEGOptions(TObject *Sender)
{
        bool isJpg;
        // 判断当前图像是否为 JPEG 图像。
        isJpg = (Image1->Picture->Graphic->ClassNameIs("TJPEGImage"));
        TJPEGImage *Jpeg;
        if (isJpg){
                // 将 Image1->Picture->Graphic 属性赋给一个 TJPEGImage 类型指针，
                // 这样才能使用 TJPEGImage 的属性和方法，对图像进行控制。
                Jpeg=(TJPEGImage *)Image1->Picture->Graphic;
                // 具体的图像控制实现。
                Jpeg->PixelFormat = TJPEGPixelFormat (PixelFormat->ItemIndex);
                Jpeg->Scale = TJPEGScale(Scale->ItemIndex);
                Jpeg->Grayscale = bool(ColorSpace->ItemIndex);
                Jpeg->Performance = TJPEGPerformance (Performance->ItemIndex);
        }
        // 决定四个组合框是否能被激活。
        Scale->Enabled = isJpg;
        PixelFormat->Enabled = isJpg;
        ColorSpace->Enabled = isJpg;
        Performance->Enabled = isJpg;
}

//---------------------------------------------------------------------------
```

4.4 实例创建分析

本章目标是完成一个名为 BCBSee32 的图像浏览程序（如图 4-5 所示）。

BCBSee32 程序以 ACDSystems 公司的著名的图像浏览程序 ACDSee32 为蓝本，支持 JPEG 图像、Bitmap 位图、Icon 图标图像、Metafile 图元、Enhanced Metafile 增强图元等多种图像格式。作为一个图像浏览工具，本程序将实现一些实用的图像浏览功能、图像转换功能以及辅助功能。

作为实现图像浏览功能的实用工具，显示图像在本程序中占了相当部分。显示图像主要使用前面讲述的 TImage 组件完成，包括了图像预览显示、图像整幅显示、全屏显示等多种方式，同时也提供了增大或减小图像大小，以及特殊的放大镜浏览功能。

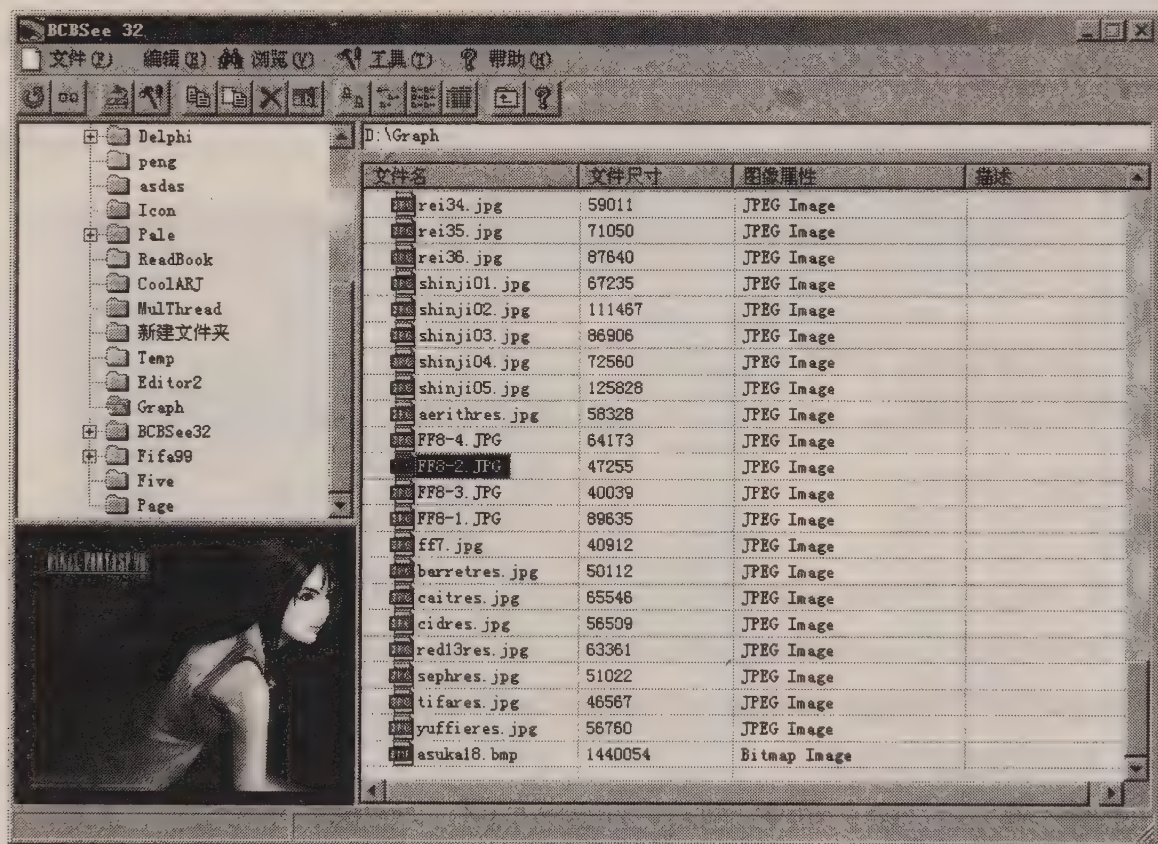


图 4-5 BCBSee32 程序——用 C++ Builder 实现的图像浏览工具

在图像格式转换方面，BCBSee32 图像浏览程序支持将各种格式的图像文件转换为 Bitmap 位图或 JPEG 格式图像，并提供专业全面的转换控制功能，例如转换为 JPEG 格式图像时可以控制 JPEG 图像的压缩比，以及 JPEG 图像的颜色深度或灰阶；转换为 Bitmap 可选 7 种不同的颜色模式。

与 ACDSee32 一致，BCBSee32 图像浏览程序分两大部分：主窗体即浏览窗体（Browser），观察窗体（Viewer）。这两部分担负着不同的功能，具体来说，在 BCBSee32 图像浏览程序的设计过程中，主要要完成以下任务：

- Win32 风格的浏览系统的实现。这与上一章所讲 BCB 文件管理器实现的方法一致，在本章里将不再进行具体讲述。
- 一些文件操作功能的实现。这部分内容仍然在上一章中实现过，在 BCBSee32 程序中可以简单的套用过来。

- 图像预览功能的实现。
- 实现观察窗体浏览。
- 实现全屏浏览。
- 增大或减小图像尺寸功能。
- 方便而又特色的放大镜功能。
- 将图像设为 Windows 墙纸。
- 图像打印功能的实现。
- 多种图像格式之间的转换的实现。

BCBSee32 工程主要包含以下部分：

- 主浏览窗体单元。类名：TForm1；文件：Unit1.cpp、Unit1.h。
- 图像显示窗体单元。类名：TviewFrm；文件：ViewFrm.cpp、ViewFrm.h。
- 转换为 JPEG 格式的控制窗体。类名：TJPGForm；文件：Jpg.cpp、Jpg.h。
- 转换为 Bitmap 格式的控制窗体。类名：TBMPForm；文件：Bmp.cpp、Bmp.h。

4.5 创建程序界面及浏览窗体部分的实现

4.5.1 创建程序界面

以下是主浏览窗体和图像显示窗体的创建要点的简要说明。

主浏览窗体主要分为四大块，如图 4-6 所示。左上是一个树视图框，用于显示磁盘的目录树结构；左下包含了一个面板组件（TPanel）和一个 TImage 图像组件，是预览显示窗

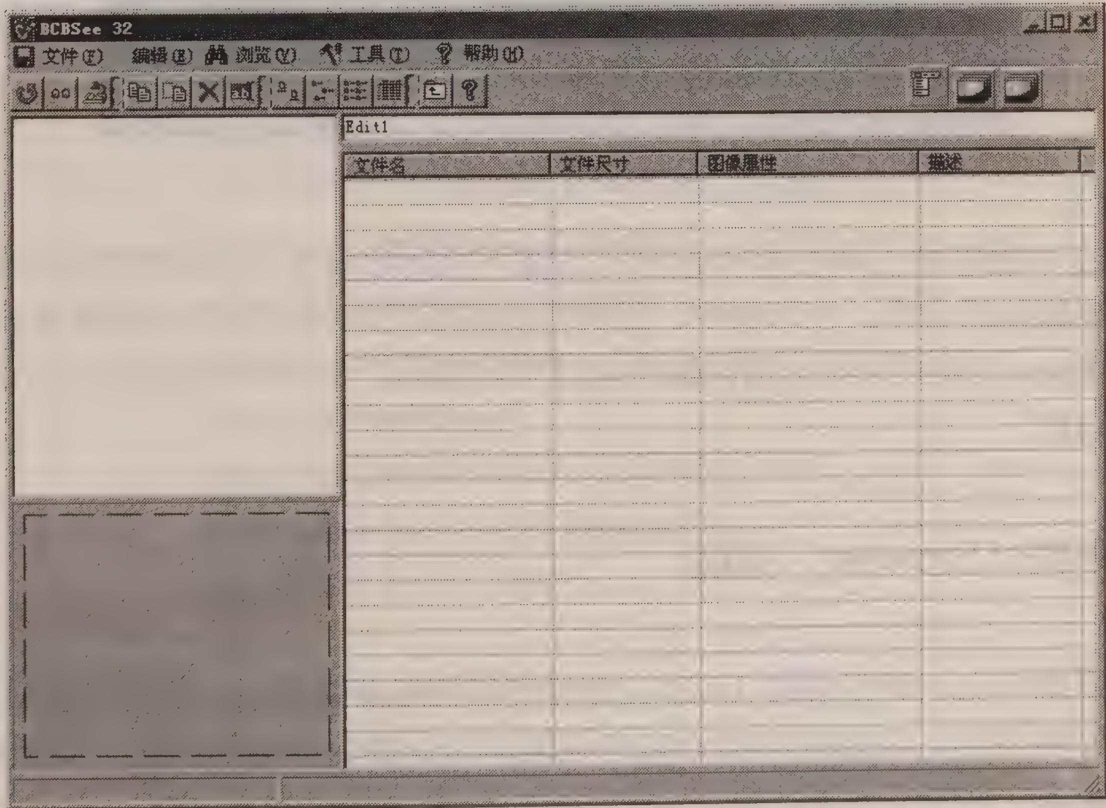


图 4-6 BCBSee32 主浏览窗体界面设计

口；右上是一个单行编辑框（TEdit），用于输入路径快捷进入；右下是一个列表视图框，用于显示出当前目录下的子目录和图像文件。

注意，如果想显示出像 ACDSee32 中带网格显的列表视图框，一定要将列表视图的 GridLine 属性设为 true。

此外主浏览窗体还包含了一个快捷工具栏，上含 13 个按钮，分别用于实现不同的功能。按钮图标由非可视组件 ImageList2 提供。主窗体另有一个状态栏，并分为两栏。

主窗体中的另一个 TImageList 组件 ImageList1 用于提供列表视图和树视图中使用的各种图标。

观察窗体的界面较为简单。它包含了主菜单，工具栏和状态栏，如图 4-7 所示。它的主要部分是两个 TImage 图像组件。其中一个用于显示，另一个位特殊处理使用。

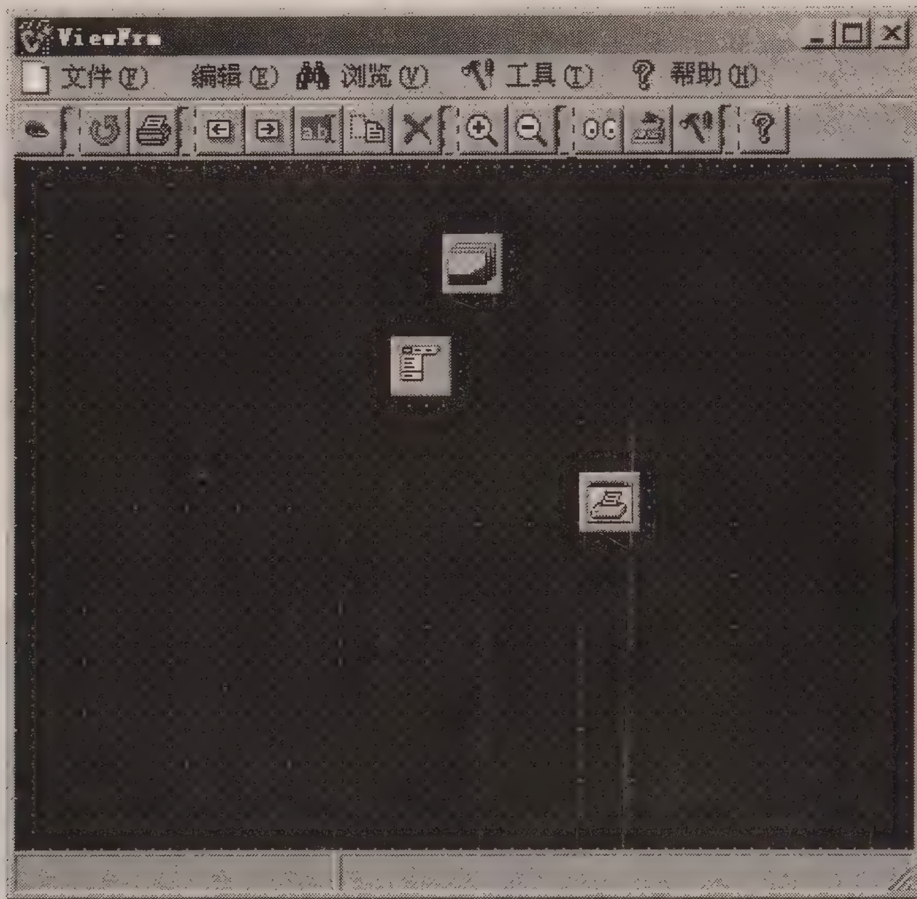


图 4-7　BCBSee32 观察窗体界面设计

另外注意在本程序中工具栏和状态栏的高度不应随意设置。因为在程序中需要由图像的大小来控制观察窗体的大小，在本例中，工具栏的高度设为 20，状态栏的高度设为 29。

观察窗体 Form 对象的 Color 颜色属性应设为 clBlack，同时观察窗体中加入打印对话框组件以实现打印功能。

4.5.2　浏览系统（Browser）实现

整个浏览系统与上一章完成的 BCB 文件管理器程序非常类似，其中树视图部分的实现完全相同，这里不再讲述。列表视图部分略有不同，这是因为在本程序中，必须判断各种图像类型，而将非图像文件的文件屏蔽掉。

实现这一功能的代码如下：

Source 列出计算机中的所有图像文件

```
//---------------------------------------------------------------------------
void TForm1::ListViewAddFile()
{
    TListItem *mItem;
    TSearchRec sr;
    AnsiString fileext;
    if (FindFirst(CurrentDir+"\\*.*",0,sr)==0){
        if (sr.Attr!=faDirectory){
            fileext=ExtractFileExt(sr.Name).LowerCase();
            if(fileext==".bmp"){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=5;
                mItem->SubItems->Add(AnsiString(sr.Size));
                mItem->SubItems->Add("Bitmap Image");
                mItem->SubItems->Add("");
            }
            else if (fileext==".jpg"||fileext==".jpeg"){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=6;
                mItem->SubItems->Add(AnsiString(sr.Size));
                mItem->SubItems->Add("JPEG Image");
                mItem->SubItems->Add("");
            }
            else if (fileext==".emf"||fileext==".wmf"){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=7;
                mItem->SubItems->Add(AnsiString(sr.Size));
                mItem->SubItems->Add("Metafiles");
                mItem->SubItems->Add("");
            }
            else if (fileext==".ico"){
                mItem=ListView1->Items->Add();
                mItem->Caption=sr.Name;
                mItem->ImageIndex=8;
```

```
            mItem->SubItems->Add(AnsiString(sr.Size));
            mItem->SubItems->Add("ICONfiles");
            mItem->SubItems->Add("");
        }
    }
    while (FindNext(sr)==0){
    if (sr.Attr!=faDirectory){
        fileext=ExtractFileExt(sr.Name).LowerCase();
        if(fileext==".bmp"){
            mItem=ListView1->Items->Add();
            mItem->Caption=sr.Name;
            mItem->ImageIndex=5;
            mItem->SubItems->Add(AnsiString(sr.Size));
            mItem->SubItems->Add("Bitmap Image");
            mItem->SubItems->Add("");
        }
        else if (fileext==".jpg"||fileext==".jpeg"){
            mItem=ListView1->Items->Add();
            mItem->Caption=sr.Name;
            mItem->ImageIndex=6;
            mItem->SubItems->Add(AnsiString(sr.Size));
            mItem->SubItems->Add("JPEG Image");
            mItem->SubItems->Add("");
        }
        else if (fileext==".emf"||fileext==".wmf"){
            mItem=ListView1->Items->Add();
            mItem->Caption=sr.Name;
            mItem->ImageIndex=7;
            mItem->SubItems->Add(AnsiString(sr.Size));
            mItem->SubItems->Add("Metafiles");
            mItem->SubItems->Add("");
        }
        else if (fileext==".ico"){
            mItem=ListView1->Items->Add();
            mItem->Caption=sr.Name;
            mItem->ImageIndex=8;
            mItem->SubItems->Add(AnsiString(sr.Size));
            mItem->SubItems->Add("ICONfiles");
            mItem->SubItems->Add("");
        }
```

```
            }
        }
    }
    FindClose(sr);
}
//---------------------------------------------------------------------------
```

这里采用了分离文件扩展名并进行判断的方法。

另外，本程序实现一个用于直接输入目录的单行文本编辑框，通过它用户可以快速进入某一目录。

应该响应 TEdit 组件的 OnKeyDown 事件。

```
void __fastcall TForm1::Edit1KeyDown(TObject *Sender, WORD &Key,TShiftState Shift)
{
    if (Key==VK_RETURN) // 当用户按下 Enter 键时，接受输入
    {
        CurrentDir=Edit1->Text;
        UpdateListView();
    }
}
```

4.5.3 预览显示处理

如同 ACDSee32，当用户选到相应的图像文件时，在左下方的预览框中应给予显示。因为预览显示框大小有限，所以当图像稍大时，需要对图像按比例拉伸显示。这需要根据图像原尺寸算出按比例拉伸后的图像的宽和高，然后进行拉伸处理。

此外，还需要将预览图像显示在预览框的中心位置。根据预览图像的宽、高以及预览框的宽、高做简单计算即可。

响应视图列表框的 OnSelectItem 事件，随时进行图像的预览显示。

Source 预览显示处理的实现

```
//---------------------------------------------------------------------------
void __fastcall TForm1::ListView1SelectItem(TObject *Sender,
    TListItem *Item, bool Selected)
{
    if (Selected&&ListView1->Selected->ImageIndex!=1){
        TImage *tmpImage=new TImage(NULL);
        tmpImage->AutoSize=true;
        tmpImage->Picture->LoadFromFile(CurrentDir+"\\"
```

```
            +ListView1->Selected->Caption);
        double pWidth=tmpImage->Picture->Width;
        double pHeight=tmpImage->Picture->Height;
        double Ratio = double(pWidth)/pHeight;
        if (pWidth > 208){
            pWidth = 208;
            pHeight= 208 / Ratio;
        }
        if (pHeight > 166){
            pHeight = 166;
            pWidth = pHeight*Ratio;
        }
        PreviewImage->Width=pWidth;
        PreviewImage->Height=pHeight;
        PreviewImage->Left=(223-pWidth)/2;
        PreviewImage->Top=(183-pHeight)/2;
        PreviewImage->Stretch=true;
        PreviewImage->Picture->LoadFromFile(CurrentDir+"\\"
            +ListView1->Selected->Caption);
        delete tmpImage;
        if (Panel1->Color!=clBlack)
            Panel1->Color=clBlack;
    }
}
//---------------------------------------------------------------------------
```

此处借用一个临时 TImage 对象导入目标图像并获得图像的原宽和原高，而后根据图像的宽和高确定预览图像的宽、高及显示位置。

4.6 实现观察窗体部分

4.6.1 为图像量身定做窗体

总的来说，观察窗体应设计为能够刚刚好容纳图像。但是又有一些例外，例如窗体必须保证一个最小宽度，使得观察窗体的工具栏能够完整的显示出来；另一方面，如果窗体高度太低，也不太好看，所以程序同样要限制窗体的最小高度。

当窗体的高度或宽度不与图像的尺寸相一致时，需要处理的另一问题是如何将图像放入窗体的正中间。总之，为了显示图像，必须为图像量身定做窗体，如图 4-8 所示。

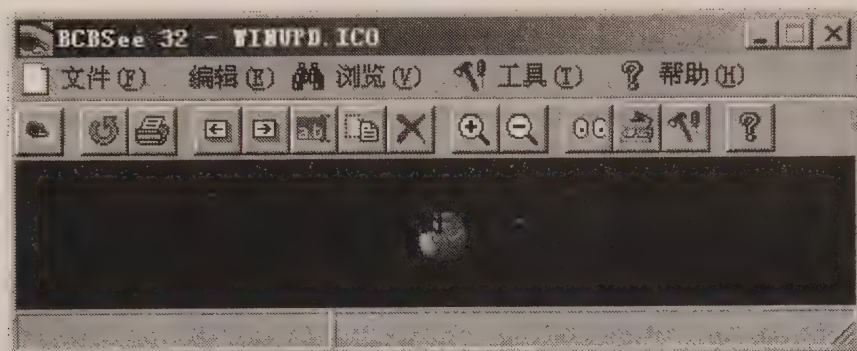


图 4-8　被显示在中间的图标图像

在主浏览窗体中定义一个用于显示观察窗体的函数 ShowPic，下面就是此函数的代码。

Source 在 Viewer 窗体中显示图像

```
//---------------------------------------------------------------------------
void __fastcall TForm1::ShowPic(int Index)
{
    ViewFrm->Image1->Picture->LoadFromFile
        (Form1->CurrentDir+"\\"+Form1->ListView1->Items->Item[Index]->Caption);
    ViewFrm->BKImage->Picture->LoadFromFile
        (Form1->CurrentDir+"\\"+Form1->ListView1->Items->Item[Index]->Caption);
    if(ViewFrm->Image1->Width<400)
    {
        ViewFrm->Image1->Left=(400-ViewFrm->Image1->Width)/2;
        ViewFrm->BKImage->Left=(400-ViewFrm->BKImage->Width)/2;
        ViewFrm->ClientWidth=400;
    }
    else {
        ViewFrm->ClientWidth=ViewFrm->Image1->Width;
        ViewFrm->Image1->Left=0;
        ViewFrm->BKImage->Left=0;
    }
    if (ViewFrm->Image1->Height<70){
        ViewFrm->ClientHeight=120;
        ViewFrm->Image1->Top=(70-ViewFrm->Image1->Height)/2+30;
        ViewFrm->BKImage->Top=(70-ViewFrm->BKImage->Height)/2+30;
    }
    else {
        ViewFrm->ClientHeight=ViewFrm->Image1->Height+50;
        ViewFrm->Image1->Top=30;
        ViewFrm->BKImage->Top=30;
```

```
        }
        // 将文件名在窗体标题上标出
        ViewFrm->Caption = Form1->Caption + " - " +
              Form1->ListView1->Items->Item[Index]->Caption;
        ViewFrm->Show();
    }
    //---------------------------------------------------------------------------
```

这里分图像在限定范围内和限定范围外两种情况，通过计算确定窗体的大小。需要注意的是，应该考虑工具栏的高度（30 像素）和状态栏的高度（20 像素）的影响。

4.6.2 Viewer 窗体中的图像浏览、幻灯功能

Viewer 窗体中的图像浏览指的是类似于 ACDSee 中用工具栏上的“⇦”和“⇨”两个按钮，或者直接使用键盘的 PageUp\PageDown 或 Up\Down 或 Left\Right 几组按键实现同一目录下所有图像的浏览功能。

这里没有复杂的技巧，需要做的仅仅是在浏览窗体单元与主窗体对象进行通信，通过 ListView 列表视图取得当前目录下所有图像的序列。

为了实现上述想法，首先为 TViewFrm 类定义一个变量（属性）Index，用于表明当前显示图像在主窗体列表视图中的位置，并在观察窗体的 OnShow 事件响应函数中初始化这个变量。

在 FormShow 函数中加入：

```
Index=Form1->ListView1->Selected->Index;
```

在本例中，Viewer 窗体的所有初始化工作都应放在 OnShow 事件响应函数中，并不是在 OnCreate 的响应函数中。因为 Viewer 窗体总是由主窗体的调用 Show 方法启动。

单击工具栏“⇦”按钮的响应函数为：

```
void __fastcall TViewFrm::ToolButton13Click(TObject *Sender)
{
    if (Index>0){
        if(Form1->ListView1->Items->Item[Index-1]->ImageIndex!=1){
            Index--;
            Form1->ShowPic(Index);
        }
    }
}
```

按下“⇦”按钮将显示上一个图像，但如果遇到目录或为列表顶端则停止。

类似的，“⇨”按钮的响应函数是：

```
void __fastcall TViewFrm::ToolButton14Click(TObject *Sender)
{
    if (Index+1<Form1->ListView1->Items->Count){
```

```
            Index++;
            Form1->ShowPic(Index);
        }
}
```

接着实现为用户使用键盘浏览图像提供支持，填写 ViewFrm 的 OnKeyDown 事件的响应函数：

```
void __fastcall TViewFrm::FormKeyDown(TObject *Sender, WORD &Key, TShiftState Shift)
{
    if (isFull) Recover(); // 用于全屏浏览，稍后讲述
    else if( Key==VK_PRIOR||Key==VK_LEFT||Key==VK_UP)
        ToolButton13Click(NULL); // 前一张图片
    else if(Key==VK_NEXT||Key==VK_RIGHT||Key==VK_DOWN)
        ToolButton14Click(NULL); // 后一张图片
}
```

这样，与 ACDSee32 相同的浏览模式就实现了。

还可以加入与 ACDSee32 类似的幻灯功能。这太容易了，仅仅需要加入一个定时器组件（TTimer），间隔一段时间执行 ToolButton13Click 和 ToolButton14Click 就可以了，有兴趣的读者可以自行完成。

4.6.3 全屏显示和放大、缩小显示

在 Windows 编程中，可以通过设置特殊的窗口状态实现全屏显示。在 C++ Builder 中，可以通过设置 TForm 类的 BorderStyle 属性和 WindowState 属性来实现这个目的。

首先请看程序中实现全屏浏览的函数。

```
void __fastcall TViewFrm::ToolButton3Click(TObject *Sender)
{
    BorderStyle=bsNone;
    WindowState=wsMaximized;
    isFull=true;
    Menu=NULL;
    ToolBar1->Visible=false;
    StatusBar1->Visible=false;
    ViewFrm->Show();
    Image1->Left=(Width-Image1->Width)/2;
    BKImage->Left=(Width-BKImage->Width)/2;
    Image1->Top=(Height-Image1->Height)/2;
    BKImage->Top=(Height-BKImage->Height)/2;
}
```

将 BorderStyle 属性设为 bsNone 可以关掉标题栏和边框，同时关掉工具栏和状态栏，再将

WindowState 属性设为 wsMaximized 就实现了全屏显示。在全屏显示后应该重新设置图像位置。

另一方面，定义一个 Recover 函数用于恢复显示。

```
void TViewFrm::Recover()
{
        BorderStyle=bsSingle;
        WindowState=wsNormal;
        IsFull=false;
        Menu=MainMenu3;
        ToolBar1->Visible=true;
        StatusBar1->Visible=true;
        Form1->ShowPic(Index);
}
```

其中 Menu=MainMenu3 一行是必要的，否则恢复尺寸后原来的菜单将不会显示出来。

实现放大缩小显示可以直接设置 TImage 类的 Stretch 属性，这在前面已有演示。但是这种方法并不能放缩图标（Icon）图像，所以在此需要采取另一种方法。

这里实现放缩的原理是，创建一个内存位图，并利用 TCanvas 类的 StretchDraw 方法将图像放大后拷贝在内存位图中，然后再通过 Graphic 属性将放大后的位图显示出来。关于 StretchDraw 方法的具体介绍请参看第 5 章。ZoomSize 是程序定义的放缩倍数常量。

以下是放大显示的函数：

Source 实现不同格式图像的放大显示

```
//---------------------------------------------------------------------------
void __fastcall TViewFrm::ToolButton8Click(TObject *Sender)
{
        Graphics::TBitmap *tmpBitmap = new Graphics::TBitmap();
        tmpBitmap->Width = Image1->Picture->Width*ZoomSize;
        tmpBitmap->Height = Image1->Picture->Height*ZoomSize;
        tmpBitmap->Canvas->StretchDraw
                (TRect(0,0,tmpBitmap->Width,tmpBitmap->Height),
        Image1->Picture->Graphic);
        BKImage->Picture->Graphic=tmpBitmap;
        Image1->Picture->Graphic=tmpBitmap;
        delete tmpBitmap;
        if(Image1->Width<400){
                Image1->Left=(400-Image1->Width)/2;
                BKImage->Left=(400-BKImage->Width)/2;
                ClientWidth=400;
```

```
        }
        else {
            ClientWidth=Image1->Width;
            Image1->Left=0;
            BKImage->Left=0;
        }
        if (ViewFrm->Image1->Height<70){
            ClientHeight=120;
            Image1->Top=(70-Image1->Height)/2+30;
            BKImage->Top=(70-BKImage->Height)/2+30;
        }
        else {
            ClientHeight=Image1->Height+50;
            Image1->Top=30;
            BKImage->Top=30;
        }
        Show();
    }
    //---------------------------------------------------------------------------
```

4.7 图像格式转换和图像打印

4.7.1 将图像转换为 Bitmap 格式

如何将一种非 Bitmap 格式的图像转换为 Bitmap 格式图像？你可能会想到使用 TBitmap 类实现 Assign 方法。但实际上，这并非通用的做法，因为 TIcon 和 TMetafile 格式的图像并不能透过 Assign 赋给 TBitmap 对象。

任何一种格式的图像都是继承于 TGraphic 类的（包括用户自定义的图像类型，也必须从 TGraphic 类派生），而 TGraphic 对象可以用 TCanvas 类的 Draw 或 StretchDraw 方法画到一个画布上。另一方面，TBitmap 类拥有一个画布类属性 Canvas，通过 Canvas 属性可以将任何一种类型的图像画到一个 TBitmap 类的实例上，然后使用 TBitmap 类的 SaveToFile 方法存储到磁盘。这样，就完成了将任何一种类型图像转换为 Bitmap 格式图像的任务。

以上是实现 Bitmap 格式转换的基本原理，这种方法支持任意图像格式到位图的转换（包括用户自定义图像）。实际中，还应该为图像转换提供更多的控制，对于位图格式转换来说，一般应提供给用户颜色深度的选择。

为此首先做出一个用于转换位图控制的对话框，它包含了一个 RadioGroup，用于选择 7 种颜色深度，如图 4-9 所示。

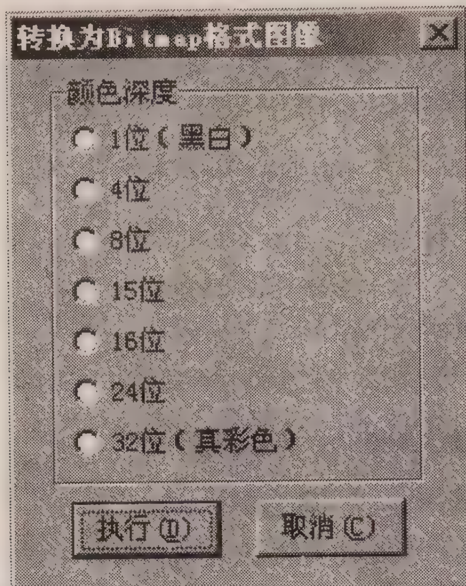


图 4-9　转换位图对话框

具体转换位图的代码如下。

Source 任意图像格式转换为位图格式的算法

```
//---------------------------------------------------------------------------
void __fastcall TBMPForm::Button1Click(TObject *Sender)
{
    SaveDialog1->Filter = GraphicFilter(__classid(Graphics::TBitmap));
    SaveDialog1->Title="另存为 Bitmap 格式图像";
    if (SaveDialog1->Execute()){
        TImage *tmpImage=new TImage(Form1);
        tmpImage->AutoSize=true;
        tmpImage->Picture->LoadFromFile (Form1->CurrentDir+"\\"
            +Form1->ListView1->Selected->Caption);
        Graphics::TBitmap *tmpBitmap = new Graphics::TBitmap();
        tmpBitmap->Width=tmpImage->Width;
        tmpBitmap->Height=tmpImage->Height;
        // 控制像素色深
        tmpBitmap->PixelFormat = TPixelFormat(RadioGroup1->ItemIndex);
        tmpBitmap->Canvas->StretchDraw(Rect(0,0,
            tmpBitmap->Width,tmpBitmap->Height),
            tmpImage->Picture->Graphic);
        tmpBitmap->SaveToFile(SaveDialog1->FileName);
        delete tmpBitmap;
        delete tmpImage;
        Close();
    }
```

```
}
//---------------------------------------------------------------------------
```

其中实现像素色深控制利用了 TBitmap 类的 PixelFormat 属性。

4.7.2 将图像转换为 JPEG 格式

由于位图的某些特殊性，将各种格式图像转换为 Bitmap 格式是容易的，而且是较为自然的。但将各种格式的图像转换为 JPEG 格式，就需要进行特殊处理。

因为 JPEG 是一种有损压缩的图像格式，所以将其他图像格式转换为 JPEG 格式一定要经过一个压缩的过程。TJPEGImage 类的 Compress 方法为进行静态图像压缩提供了支持。

在实际中，需要通过 Bitmap 格式作为媒介，即首先将各种格式的图像文件转换为 Bitmap 格式，这可以利用上节所示的技术实现。然后，需要使用 TJPEGImage 的 Assign 方法，Assign 方法可以将一个位图图像数据赋给一个 TJPEGImage 对象，然后再利用 TJPEGImage 类的 Compress 方法进行压缩，最后使用 TJPEGImage 类的 SaveToFile 方法，就完成了到 JPEG 格式的转换工作。

当然，进行 JPEG 格式转换时，也应该给用户提供各种控制。和一些专业的图像软件一致，在 BCBSee32 中提供了灰阶（Grayscale）、像素色深、压缩质量三方面的控制。为此应首先设计一个用于与用户交互的对话框，如图 4-10 所示。

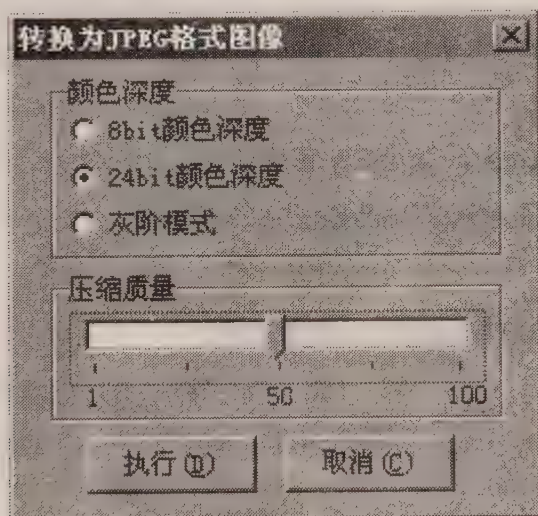


图 4-10　转换为 JPEG 格式图像的设置对话框

具体转换为 JPEG 格式图像的代码如下。

Source 任意图像格式转换为 JPEG 格式的算法

```
//---------------------------------------------------------------------------
void __fastcall TJPGForm::Button1Click(TObject *Sender)
{
    SaveDialog1->Filter = GraphicFilter(__classid(TJPEGImage));
    SaveDialog1->Title="另存为 JPEG 格式图像";
    if (SaveDialog1->Execute()){
```

```
        TImage *tmpImage=new TImage(Form1);
        Graphics::TBitmap *tmpBitmap = new Graphics::TBitmap();
        TJPEGImage *Jpeg =new TJPEGImage();
        tmpImage->AutoSize=true;
        tmpImage->Picture->LoadFromFile (Form1->CurrentDir+"\\"
            +Form1->ListView1->Selected->Caption);
        tmpBitmap->Width=tmpImage->Width;
        tmpBitmap->Height=tmpImage->Height;
        tmpBitmap->Canvas->StretchDraw(Rect(0,0,
            tmpBitmap->Width,tmpBitmap->Height),
        tmpImage->Picture->Graphic);
        switch (RadioGroup1->ItemIndex){
        case 0:
            Jpeg->PixelFormat=jf8Bit;
            break;
        case 1:
            Jpeg->PixelFormat=jf24Bit;
            break;
        case 2:
            Jpeg->Grayscale=true;
        }
        Jpeg->Assign((TPersistent *)tmpBitmap);
        Jpeg->CompressionQuality=TrackBar1->Position;
        Jpeg->Compress();
        Jpeg->SaveToFile(SaveDialog1->FileName);
        delete tmpBitmap;
        delete tmpImage;
        delete Jpeg;
        Close();
    }
}
//---------------------------------------------------------------------------
```

在进行灰阶（Grayscale）、像素色深、压缩质量的控制时，分别利用了 TJPEGImage 类的 Grayscale、PixelFormat、CompressionQuality 属性。但一定要注意，设置 Grayscale 和 PixelFormat 属性一定要在使用 Assign 方法之前，否则将不能够正常转换。

4.7.3 图像打印输出

图像输出设备在一般情况下是显示器屏幕，但是打印机也是一种非常常用的图像输出设备。本节将实现将图像输出到打印机的功能。

C++ Builder 提供了 TPrinter 类用于实现打印功能，TPrinter 封装了 Windows 打印机接口。TPrinter 类在实例化时就完成了必要的初始化工作，在开始进行打印操作时，应使用 TPrinter 类的 BeginDoc 方法，在结束一项打印作业时，应使用 EndDoc 方法。

Printer 函数用于获得全局 TPrinter 类的对象，TPrinter 类也有一个 TCanvas 画布类属性 Canvas，Canvas 属性用于描述打印机当前打印页的作图表面。因此，进行图像的打印输出，与控制一个 Canvas 画布、将图像画在画布上是基本一致的。

在 BCBSee32 程序中，图像打印代码如下：

Source 实现图像打印功能的相关代码

```
//---------------------------------------------------------------------------
void __fastcall TViewFrm::ToolButton17Click(TObject *Sender)
{
      float AspectRatio,OutputWidth, OutputHeight;
      if (!PrintDialog1->Execute()) return;
      Printer()->BeginDoc();
      OutputWidth = Image1->Picture->Width;
      OutputHeight = Image1->Picture->Height;
      Ratio = OutputWidth / OutputHeight;
      if (OutputWidth < Printer()->PageWidth&&OutputHeight < Printer()  ->PageHeight){
            if (OutputWidth < OutputHeight){
                  OutputHeight = Printer()->PageHeight;
                  OutputWidth = OutputHeight * Ratio;
            }
            else{
                  OutputWidth = Printer()->PageWidth;
                  OutputHeight = OutputWidth / Ratio;
            }
      }
      if (OutputWidth > Printer()->PageWidth){
            OutputWidth = Printer()->PageWidth;
            OutputHeight= OutputWidth / AspectRatio;
      }
      if (OutputHeight > Printer()->PageHeight){
            OutputHeight = Printer()->PageHeight;
            OutputWidth = OutputHeight * Ratio;
      }
      Printer()->Canvas->StretchDraw(Rect(0,0,
            floor(OutputWidth), floor(OutputHeight)),
       Image1->Picture->Graphic);
```

```
}
//---------------------------------------------------------------------------
```

此函数在观察窗体（Viewer）中实现。在预览显示时，本程序采用了简单拉伸处理，在此处则采取按比例放缩图像使图像适合打印纸张尺寸。为实现此目的必须按照图像的尺寸和纸张的尺寸计算出一个合适的放缩比例，前提是保证图像的高宽之比不变，Ratio 变量即表示图像的高宽比。找到合适的输出尺寸后，使用 StretchDraw 将图像输出到打印机上。

最后要说的是 TPrintDialog 打印对话框组件，它是 C++ Builder 提供的标准对话框，激活它能够让用户进行打印前的必要设置。

4.8 实现特色功能

BCBSee32 程序中包含了两项特色功能，其一是设置墙纸，其二是观察窗体中提供的放大镜功能。

设置墙纸功能在 ACDSee32 中同样提供，但放大镜功能则是一个有特色而且很有趣的设计。

4.8.1 设置墙纸

完成一个设置墙纸任务，需要做以下两件事：

- 将图像转换为位图格式，因为一般 Windows 系统只接受位图格式图像作为墙纸。
- 将转换好的位图设置成当前墙纸。

设置墙纸的关键是调用 SystemParametersInfo API 函数，具体程序如下：

Source 将图像设置为 Windows 墙纸

```
//---------------------------------------------------------------------------
void __fastcall TForm1::SetWallPaper()
{
    Graphics::TBitmap *tmpBitmap = new Graphics::TBitmap();
    tmpBitmap->Width=ViewFrm->Image1->Width;
    tmpBitmap->Height=ViewFrm->Image1->Height;
    tmpBitmap->Canvas->StretchDraw (
        Rect(0,0,tmpBitmap->Width,tmpBitmap->Height),
        ViewFrm->Image1->Picture->Graphic);
    char dir[50];
    GetWindowsDirectory(dir,50);
    tmpBitmap->SaveToFile(AnsiString(dir) + "\\BCBSeeWallPaper.bmp");
    SystemParametersInfo(SPI_SETDESKWALLPAPER, 0,
```

```
        (AnsiString(dir)+"\\BCBSeeWallPaper.bmp").c_str(), 0);
}
//---------------------------------------------------------------------------
```

将图像转换为位图的实现在前面已有详细讲解，在此不再赘述。转换完成的图像应该存入一个特定的目录，Windows 根目录是一个不错的选择。这里使用 GetWindowsDirectory 函数获取本地计算机的 Windows 目录，这个函数定义如下：

```
UINT GetWindowsDirectory(
  LPTSTR lpBuffer,   // 返回 Windows 目录的字符串的地址
  UINT uSize   // 返回 Windows 目录的字符串的大小
 );
```

设置墙纸使用 SystemParametersInfo 函数。该函数的功能相当强大，它用于设置或查询各种系统参数，函数定义如下：

```
BOOL SystemParametersInfo(
  UINT uiAction,// 查询或设置哪一个系统参数
  UINT uiParam,       // 取决于 uiAction 的值
  PVOID pvParam,   // 取决于 uiAction 的值
  UINT fWinIni  // 决定是否更新 Win.ini 配置文件，是否将设置改变消息立即发出。
 );
```

uiAction 参数具体决定函数执行的功能，这个参数取值有 100 个左右，限于篇幅在此不能一一讲述，仅举常用两例：

SPI_SETDESKWALLPAPER	本程序中使用的值，用于设置 Windows 墙纸。
SPI_SCREENSAVERRUNNING	屏蔽系统键。使得所有系统键（如 Alt+Enter、Ctrl+Alt+Del、Ctrl+Esc 等）在你的程序中失效。

4.8.2 放大镜的实现

放大镜是本程序的一个有趣的设计，它提供给用户局部放大浏览功能。当用户的鼠标向上移动时，鼠标所在的一块区域被放大显示，其效果如图 4-11 所示。



图 4-11 局部放大浏览

放大镜实现的原理并不复杂，它使用了 TCanvas 的 CopyRect 方法，其功能是将源画布上的一个指定矩形区域内的像素，拷贝到目的画布上的一个指定矩形区域中。

读者可能已经注意到，在本程序观察窗体实现浏览时，图像组件有两个——Image1 和 BKImage，它们的内容完全相同。其实，BKImage 图像组件的加入就是为实现放大镜功能做铺垫。

具体的实现方法如下：

（1）在按钮单击响应函数中做一些必要的初始化工作。主要是为提供对 JPEG 格式图像的支持，因为如果 TImage 中装入的是一个 TJPEGImage 格式图像时，是不能对 Canvas 画布属性进行操作的。

```
void __fastcall TViewFrm::ToolButton12Click(TObject *Sender)
{
    isZoom=true;
    doHide=true;
    if (Image1->Picture->Graphic->ClassNameIs("TJPEGImage")){
        Graphics::TBitmap *tmpBitmap = new Graphics::TBitmap();
        tmpBitmap->Width = Image1->Picture->Width;
        tmpBitmap->Height = Image1->Picture->Height;
        tmpBitmap->Canvas->StretchDraw
            (TRect(0,0,tmpBitmap->Width,tmpBitmap->Height),
        Image1->Picture->Graphic);
        Image1->Picture->Graphic = tmpBitmap;
        BKImage->Picture->Graphic=tmpBitmap;
        delete tmpBitmap;
    }
}
```

（2）编写一个 ZoomIt 函数，用于处理图像的放大和恢复，当移动鼠标时调用。这是实现图像局部放大最重要的过程，其源代码如下：

```
void TViewFrm::ZoomIt(int CenterX, int CenterY, int Size)
{
    SourceRect.Left=CenterX-Size;
    SourceRect.Top=CenterY-Size;
    SourceRect.Right=CenterX+Size;
    SourceRect.Bottom=CenterY+Size;
    DestRect.Left=CenterX-dSize;
    DestRect.Top=CenterY-dSize;
    DestRect.Right=CenterX+dSize;
    DestRect.Bottom=CenterY+dSize;
    Image1->Canvas->CopyRect(DestRect,
    BKImage->Canvas, SourceRect);
}
```

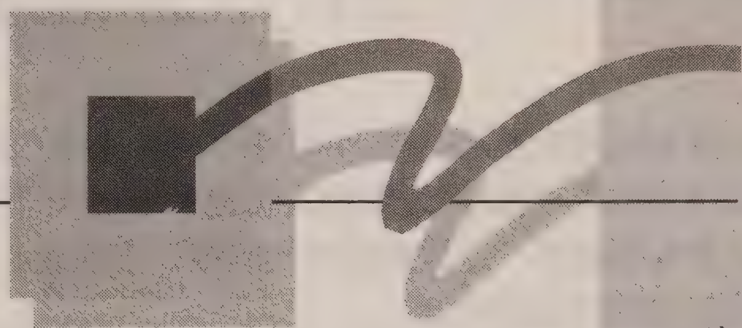
（3）响应 Image1 的 OnMouseMove 事件，当鼠标在图像上移动时，获取其位置，并作为调用 ZoomIt 函数的参数。此时，需要将光标隐藏，并使“放大镜”出现。随着“放大镜”的移动，图像新的部位被放大，滑过的部位又恢复原状。

```
void __fastcall TViewFrm::Image1MouseMove(TObject *Sender,
    TShiftState Shift, int X, int Y)
{
      if (isZoom){
            NewX=X;
            NewY=Y;
            Image1->Visible=false;
            if (doHide){
                  OldX=NewX;
                  OldY=NewY;
                  doHide=false;
                  ShowCursor(false);
            }
            else
                  ZoomIt(OldX, OldY, dSize);
            ZoomIt(NewX, NewY, sSize);
            Image1->Visible=true;
            OldX=NewX;
            OldY=NewY;
      }
}
```

第5章

图像编辑软件 BCB 画板
——数字图像处理和图像编辑

画布类 TCanvas 及相关内容
图像绘制软件的实现
快速数字图像处理技巧
图像的色彩调整及滤镜技术



本章导读

图像绘制和图像数字处理是图形图像编程的重要方面。本章将通过一个类似 Windows 95/98 画图工具（MSPaint）的范例程序——BCB 画板，讲述这一方面的编程技术。BCB 画板具有和 MSPaint 相同的界面，但功能更为强大——BCB 画板程序在实现完整绘图和图像编辑功能的同时，实现了许多 Photoshop 中的色彩调整功能和滤镜功能。在对这个精彩范例程序的分析和创建过程中，读者将能了解到很多有关图形图像编程的技术和技巧。

本章内容包括:

- C++ Builder 图形图像编程的基础——TCanvas 类的介绍。同时，将介绍与 TCanvas 类密切相关的 TPen 画笔类，TBrush 画刷类以及 TColor 类型。
- “BCB 画板”程序图像绘制，图像编辑功能的实现。这涉及了相当多的内容，包括画笔工具、画刷工具、颜料桶工具、橡皮工具、喷枪工具的实现，工具箱和颜料盒的实现，规则图形的绘制，在图像上绘制文字，图像的放大和缩小，图像旋转，区域选择和剪贴板功能的实现，图像打印，反色和图像尺寸控制等等。
- 高级图像处理。将讲述包括图像的快速处理，色彩调整技术（亮度调整、对比度调整、灰阶、色彩均衡），滤镜技术（模糊、锐化、雕刻）等。

5.1 TCanvas 画布类

TCanvas 画布类封装了 Windows GDI，以及相关的图像绘制和控制函数。TCanvas 类简化了很多繁琐的操作，同时也为编程人员提供了一个非常灵活的绘图场所，使编程者可以在具有 Canvas 属性的组件（这样的组件相当多）表面随心所欲地绘图。这对完善用户界面或者制作一些屏幕特技都有重要作用。

5.1.1 TCanvas 类的重要属性和方法

TCanvas 类对象常作为其他组件的属性出现，用于在运行阶段绘制图形，或进行图像的访问和控制。

1. TCanvas 类重要属性

- Pen 属性。

声明：__property TPen* Pen = {read=FPen, write=SetPen};

该属性决定画布对象的当前画笔，包括绘图位置，表现绘图时的颜色、粗细等状态。

- Brush 属性。

声明：__property TBrush* Brush = {read=FBrush, write=SetBrush};

该属性决定画布对象的当前画刷，决定绘制形体的填充效果。

- Pixels 属性。

声明：__property TColor Pixels[int X][int Y] = {read=GetPixel, write=SetPixel};

描述画布上每一像素的颜色值，是一个 TColor 类型的二维数组。

- ClipRect 属性。

声明：__property Windows::TRect ClipRect = {read=GetClipRect};

裁剪区域，用于设定画布上的画图区域，在裁剪区域之外的画图操作都没有效果。

- Handle 属性。

声明：__property HDC Handle = {read=GetHandle, write=SetHandle, nodefault};

WindowsGDI 句柄，访问 Handle 属性相当于执行 GetDC 函数。

- Font 属性。

声明：__property TFont* Font = {read=FFont, write=SetFont};

描述当前画布的字体类型。与 TextOut 方法相关。

2. TCanvas 类重要方法

相比之下，TCanvas 类的方法具有更重要的意义，在 C++ Builder 中所有绘图操作都要通过 TCanvas 类的方法完成，下面列举 TCanvas 类的重要方法。

- Arc 方法。

声明：void __fastcall Arc(int X1, int Y1, int X2, int Y2, int X3, int Y3, int X4, int Y4);

Arc 方法在椭圆上画一段弧，椭圆由（x1，y1）、（x2，y2）两点所确定的椭圆所决定，弧的起点是椭圆圆周和椭圆中心与（x3，y3）连线的交点。弧矩形终点是椭圆圆周和椭圆中心与（x4，y4）连线的交点，以逆时针方向画弧。

- Chord 方法。

声明：void __fastcall Chord(int X1, int Y1, int X2, int Y2, int X3, int Y3, int X4, int Y4);

Chord 方法连接椭圆上的两点，椭圆由（x1，y1）、（x2，y3）两点所确定的矩形决定，（x3，y3）是始点，（x4，y4）是终点。

- Brushcopy 方法。

声明：void __fastcall BrushCopy(const Windows::TRect &Dest, TBitmap* Bitmap, const Windows::TRect &Source, TColor Color);

Brushcopy 方法把位图的一部分复制到画布的某个矩形区域，并用画笔的当前颜色替换位图的颜色。参数 Dest 声明画布的一个矩形区域，该矩形用以填充位图，Bitmap 声明位图；Source 声明位图中的矩形区域，该区域上的位图将被复制；Color 声明画笔中的颜色，用以替换位图的颜色。

- CopyRect 方法。

声明：void __fastcall CopyRect(const Windows::TRect &Dest, TCanvas* Canvas, const Windows::TRect &Source);

此方法从另一个画布对象上复制部分图像到该画布。Canvas 表示源画布，Source 是源画布上要复制的图像区域，Dest 表示目标画布上将接受复制图像的矩形区域。TCanvas 类 CopyMode 属性决定复制方式。

- Draw 方法。

声明：void __fastcall Draw(int X, int Y, TGraphic* Graphic);

此方法在画布给定的像素点坐标（x，y）处画 Graphic 所给的图像。

- Ellips 方法。

声明：void __fastcall Ellipse(int X1, int Y1, int X2, int Y2);

Ellips 方法在画布指定的矩形边界上画一个椭圆，（x1，y1）是矩形左上角的像素坐标，（x2，y2）是矩形右下角的像素坐标。

- FloodFill 方法。

声明：void __fastcall FloodFill(int X, int Y, TColor Color, TFillStyle FillStyle);

FloodFill 方法用于填充一个区域。FillStyle 参数指明填充方式，有两个值：fsSurface 和 fsBorder。fsSurface 表示填充区域直到遇到与 Color 颜色不同的点，fsBorder 表示填充区域直到遇到与 Color 参数颜色相同的点。

- LineTo/ MoveTo 方法。

声明：void __fastcall LineTo/MoveTo(int X, int Y);

LineTo 方法从当前位置画一条线至（x，y）所指定的位置，并把笔的位置移至（x，y），MoveTo 将笔的当前位置设置到点（x，y）处。请注意虽然笔的当前位置在 PenPos 属性中，但改变笔的当前位置应使用 MoveTo 方法，而不要设法改变 PenPos 的值。

- Polygon 方法。

声明：void __fastcall Polygon(const Windows::TPoint * Points, const int Points_Size);

Polygon 方法在画布上绘制一系列的点，各点依次连成线，最后将首尾两点相接形成一个

区域，并用当前笔刷填充此区域。

- Polyline 方法。

声明：void __fastcall Polyline(const Windows::TPoint * Points, const int Points_Size);

Polyline 方法在画布上用当前画笔绘制一系列的点，各点依次连成线。

- PolyBezier 方法。

声明：void __fastcall PolyBezier(const Windows::TPoint * Points, const int Points_Size);

PolyBezier 方法在画布上画出一条贝塞尔曲线，Points 指向型值点数组，Points_Size 指定型值点的个数。

- StretchDraw 方法。

声明：void __fastcall StretchDraw(const Windows::TRect &Rect, TGraphic* Graphic);

StretchDraw 方法在指定的矩形内画一图像，图像延伸改变大小以适应矩形。此方法在上一章已多次用到。

- Rectangle 方法。

声明：void __fastcall Rectangle(int X1, int Y1, int X2, int Y2);

Rectangle 方法在画布上用当前画刷绘制矩形，（x1,y1）是矩形的左上角，（x2,y2）是矩形的右下角。

- DrawFocuseRect 方法。

声明：void __fastcall DrawFocusRect(const Windows::TRect &Rect);

此方法用于绘制一矩形，特殊的地方在于使用异或（XOR）模式绘制，第二次调用时原绘制矩形将消失。

- TextOut 方法。

声明：void __fastcall TextOut(int X, int Y, const AnsiString Text);

在画布上输出文字，文字字体、字型等属性由 TCanvas 类 Font 属性决定。

5.1.2 TPen、TBrush 和 TColor

1. TPen 画笔类

TPen 画笔类是 Windows 画笔对象（HPEN）的封装，用于在画布上绘制各种线段，笔的颜色由 Color 属性指定，线段宽度由 Width 属性指定，Style 属性指定线段的各种类型，如表 5-1 所列。

表 5-1 画笔类 Style 属性取值及其含义

Style	含　义
pSolod	实线段
psDash	由下划线组成的线段
psDot	由点组成的线段
psDashDotDot	双点划线
psDashDot	点划线
psClear	无显示线段
psInsideFrame	边界的矩形线框

Mode 属性定义线段的颜色模式。可结合当前的颜色、屏幕颜色或它们反转值对线段的颜色重新定义，但不改变 Color 属性，取值见表 5-2。

表 5-2 画笔类 Mode 属性取值及其含义

Mode	像素颜色
pmBlack	黑色
pmWhite	白色
pmNop	不变
pmCopy	使用 Color 属性中的颜色
pmNotCopy	笔颜色的反转值
pmMergePenNot	笔的颜色与屏幕颜色反转值的结合
pmNaskNotPen	屏幕颜色与笔颜色
pmMergeNotPen	屏幕颜色与笔颜色反转值的结合

2. TBrush 画刷类

TBrush 画刷类用以填充图形，如用画刷颜色或图案对矩形或椭圆进行填充。TBrush 是 Windows 画刷（HBRUSH）对象的封装。

TBrush 类的颜色定义在 Color 属性中。TBrush 类的另一个属性是 Bitmap，该属性可以获取或设定填充位图，但是设定的填充位图只能为 8×8 大小，如果位图大于 8×8，则只有左上角 8×8 大小有效。

Style 属性定义了画刷填充图形的风格，Style 的取值及其图案如图 5-1 所示。

图 5-1 TBrush 类的 Style 的取值及其图案

3. TColor 类型

TColor 是图形图像编程中非常重要的类型，TColor 类型用于描述一个对象的颜色。例如所有组件的 Color 属性就是 TColor 类型， 在 Graphics.hpp 头文件中 TColor 定义如下：

enum TColor {clMin=-0x7fffffff-1, clMax=0x7fffffff};

TColor 类型实际上是一个 DWORD 类型（32 位无符号整形），所以它是一个四字节数据，低 3 位字节代表 RGB 相应的颜色，如 0x00FF0000 表示纯蓝，0x0000FF00 表示纯绿，0x000000FF 表示纯红，0x00000000 表示黑色，0x00FFFFFF 表示白色等。如果最高位字节是 0x00，则表示用系统调色板中最相近的颜色；最高位字节是 0x01，则表示用当前调色板中最相近的颜色匹配；最高位字节是 0x02，则用当前设备描述表中逻辑调色板的次相近颜色匹配。

根据 TColor 类型的颜色存储结构，可以使用位运算符将 RGB 三色的值从 TColor 类型变量中分离出来，如下面这段代码：

```
void AnalysisRGB(TColor Color,BYTE &R,BYTE &G,BYTE &B)
{
        R=BYTE(Color);
        G=BYTE(Color>>8);
        B=BYTE(Color>>16);
}
```

Graphic.hpp 头文件中还定义了一些常用的颜色常量，这些常量或直接映射成系统调色板中最相近的颜色，或映射成 Windows 系统定义颜色。直接映射成系统调色板中的颜色有：clAqua、cLBlack、clBlue、clbkGrray、clFuchsoa 等等；映射成 Windows 系统定义颜色有：clBackground、clActiveCaption、clGrayText 等等。

5.1.3 重画问题

利用 TCanvas 类的方法，可以在任何一个用有画布属性的组件表面绘制图像，例如以下组件都具有 Canvas 属性：TComboBox，TDBComboBox，TStringGrid，TDBListBox，TForm，THeaderControl，TImage ，TListBox，TDirectoryListBox，TOutline，TPaintBox，TStatusBar，TStringGrid，TPrinter，TDrawGridTFileListBox 等等。

但是如果仅仅简单地将图形画在画布上，当被画组件被其他窗体覆盖后重新激活时，原有的图形或图像会消失或者变得残缺不全。又如当窗口进行最小化又重新恢复时，也会引起画布上图像消失。要解决这个问题，就必须重画。

在 C++ Builder 中，重画可以使用组件的 OnPaint 事件，OnPaint 事件在图像被破坏需要重画时被触发，所以，在 OnPaint 事件的响应函数中将图像重新绘制一遍就能使图像恢复了。

因此利用 Canvas 对象进行绘制时，一般要将绘制代码放在 OnPaint 的响应函数中，而放在程序中的其他部分都将是徒劳的。

但有两个组件是例外，对 TImage 组件和 TShape 组件的 Canvas 属性进行绘制就不需要进行重画，而是由系统自动进行。这两个组件的重画代码都被隐藏了起来，这也正是它们的优点所在。

另外，有时我们需要强迫重画窗体，这时可以使用 Invalidate 方法使窗体无效而重画。

5.2 实例创建分析

本章的实例程序——“BCB 画板”是一个图像编辑和图像处理软件。它模仿了 Windows 98/95 的画图程序的风格。在 Windows 画图程序中，所有画图命令都是用户通过左边的工具栏发出的。应该注意，在画图程序中，用户按下工具栏上的按钮后，操作并不是立刻执行，而仅仅是改变了当前的操作状态，真正的画图操作是由鼠标直接发出的。这种操作方式和 Windows 画图程序是相同的。

程序中定义了 TDrawingTool 类型和一个 TDrawingTool 类型的变量 ToolState，用于标明画图状态。真正的画图操作在 OnMouseMove、OnMouseDown 和 OnMouseUp 几个响应函数中执行，通过画图工具栏实现各种画图操作是本例的关键之一。BCB 画板程序如图 5-2 所示。

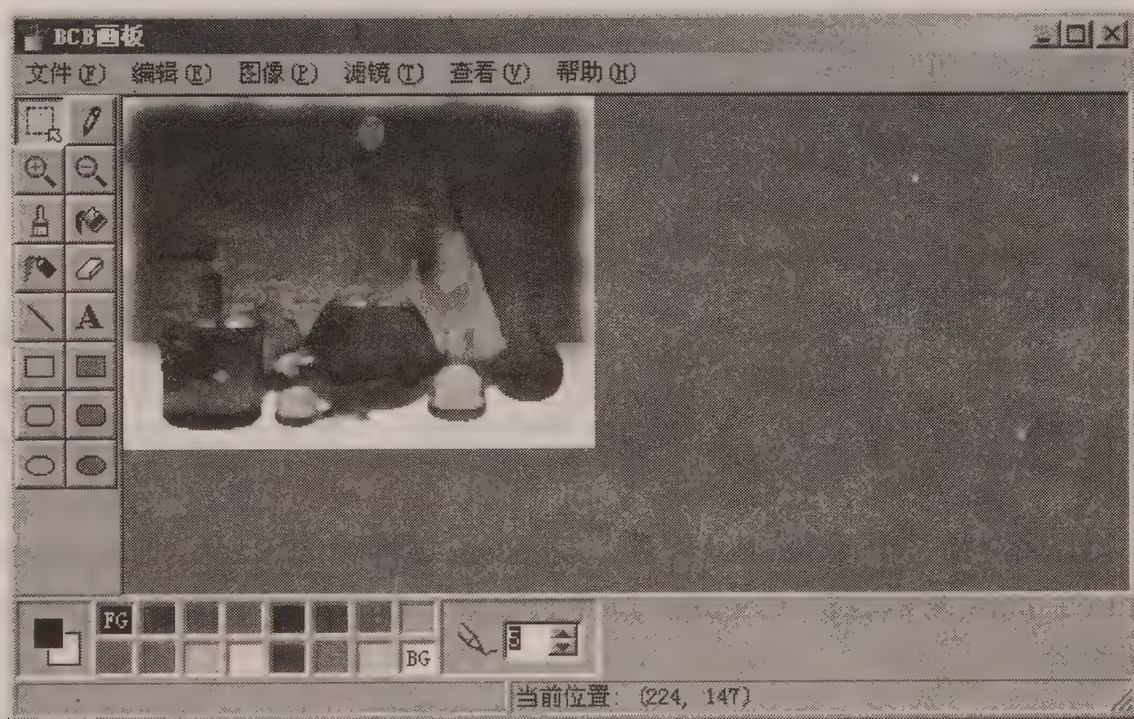


图 5-2 功能强大的 BCB 画板程序

处理利用鼠标完成画图功能是本例的另一关键，根据当前画图状态的不同，在三个鼠标响应函数中要做不同的处理。不同的画图操作具有较大区别：画笔、画刷和橡皮属于“乱画型”，按下鼠标不放移动鼠标就可以随意画图；而颜料桶、喷枪属于“点画型”；画规则图形则需要由用户确定两点，但在本程序中我们还将进一步实现橡皮筋功能。另外还有放大缩小，绘制文字等等，都要根据鼠标操作方式的不同作不同的处理。

对于图像处理功能（包括色彩调整，简单滤镜），绝大部分将在本章最后出现。完成这些功能需要逐个对像素进行操作，处理速度问题就变得很重要了。

具体来说，创建“BCB 画板”程序需要逐步完成以下工作：

- 创建应用程序界面，实现“工具箱”和“颜料盒”的外观。
- 实现使用不同的光标。
- 实现工具栏响应规则和颜料盒的相关代码。
- 具体完成画图功能。包括画笔工具、画刷工具、颜料桶工具、橡皮工具、喷枪工具

的实现，规则图形的绘制，在图像上绘制文字，图像的放大和缩小等。

- 编辑功能，图像区域选择以及剪切、复制和粘贴。
- 完成辅助功能。包括图像的新建、打开和存储，图像格式转换存储，图像打印，图像尺寸调整，图像旋转等。
- 完成图像处理功能。包括色彩调整，亮度/对比度调整，反色/灰阶，以及几个简单滤镜的实现。

BCBDraw 工程中主要包含下面的一些文件：

- 主窗体单元。类名：TMainForm，文件：Unit1.cpp，Unit1.h。
- 几个对话框窗体，包括：色彩平衡调整窗体（TCForm）、绘制字体对话框（TFontForm）、新建图像对话框（TNewBMPForm）、调整图像大小的对话框（TResizeBMPForm）、以及一个 About 窗体（TAboutBox）。

完成“BCB 画板”程序要做很多工作，但这一部分内容确实非常有趣。相信在“BCB 画板”程序的实现过程中，读者可以学到很有价值的图形图像编程技术。

5.3 图像编辑程序框架

5.3.1 创建应用程序界面

本程序的主窗体界面并不复杂，它包括图像绘制窗体、竖立的工具栏以及一个称为“颜料盒”的工具面板（Panel）组合而成。

左边的竖立工具栏是一个 TCoolBar 组件，其 Align 属性设为 alLeft，EdgeBorders 属性设为 [ebTop, ebBottom]。同时需要调整 CoolBar 组件的宽度，使工具栏宽度适合放置两个合适大小的 SpeedButton 组件。

在工具栏上添加 16 个 SpeedButton，并设置相应的图标。同时注意将所有 SpeedButton 组件的 GroupIndex 设为同一值，如 1，并设置 AllowAllUp 属性为 false，这样才能使这一组 SpeedButton 有 RadioButton 的性质，即有且只有一个按钮处于 Down 的状态。

“颜料盒”工具栏采用 TPanel 面板组件来实现。“颜料盒”面板又分为三大块，最左边是一个面板组件，在面板的适当位置放置两个 TShape 组件，用于显示颜料盒中选择的前景色和背景色，分别命名为 FGShape 和 BGShape，设置 Shape 属性为 stSquare（方形），最后还要设置两个 Shape 组件的 Brush 属性和 Pen 属性的颜色。

“颜料盒”工具栏的关键部分是一个 TCColorGrid 组件。这个组件位于选项板的 Samples 页，用于颜色可视化的选择。对这个组件设置如下：

```
GridOrdering = go8x2            // 颜色块以每行 8 个共两行的模式显示
BackgroundEnabled = true        // 支持背景色选择
BackgroundIndex = 15            // 初始背景色为白色
ForegroundEnabled = true        // 支持前景色选择
ForegroundIndex = 0             // 初始前景色为黑色
```

TCColorGrid 组件右边是一个 TPanel 面板组件，其上放置一个 Image 组件用于显示图标，

以及一个 TCSpinEdit 组件用于选择笔宽。TCSpinEdit 组件位于选项板的 Samples 页，当然完全可以使用一个 TEdit 组件和一个 TUpDown 上下组件组合完成 TCSpinEdit 组件的功能。BCB 画板主窗体界面如图 5-3 所示。

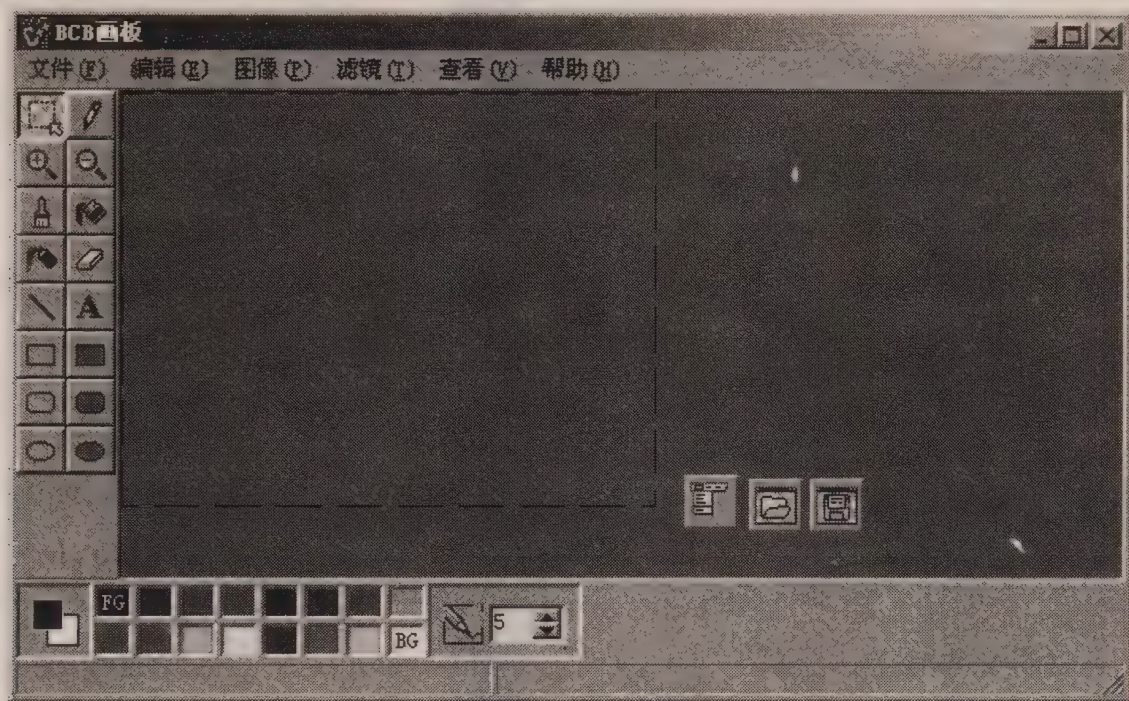


图 5-3 主窗体界面设计

窗体 Form 部分，应设置其颜色为 clGrayText。窗体上关键组件是 TImage 组件，命名为 Image，用于编辑图像的显示和处理。

此外，其他组件还包括组窗体菜单、打开对话框(OpenDialog)、存盘对话框(SaveDialog)、打印对话框(PrintDialog)等等。

5.3.2 使用光标

在绘图操作时，鼠标的光标会变成不同的形状。例如，在选择铅笔绘图时，当鼠标移动到图像上时，鼠标的光标会变成一支铅笔；在填充时，鼠标的光标会变成一个颜料桶等等，下面将讲述这种功能的实现方法。

Screen 全局对象的 Cursors 属性包含了当前可用的所有光标（系统内建光标）。同时，如果在应用程序中想使用自己的光标，也必须访问 Screen 全局对象的 Cursors 属性，进行光标创建。Cursors 属性是一个 HICON 数组，它是这样定义的：

```
__property HICON Cursors[int Index] = {read=GetCursors, write=SetCursors};
```

一般来说，在应用程序中创建和使用自己的光标有以下几个步骤：

（1）创建光标资源。可以使用 C++Builder 的 Image Editor 工具。

（2）声明一个光标常量。这个光标常量必须不与系统内建光标定义的常量冲突。图 5-4 是几个重要的系统内建光标及其定义的常量值。

（3）使用 Windows API 函数 LoadCursor（从资源中）或 LoadCursorFromFile（从文件中）获取光标句柄。

（4）通过全局对象 Screen 的 Cursors 属性设置光标。

Constant	Value	Image
crDefault	0	Whatever cursor is the default for the window class (usually crArrow).
crNone	-1	
crArrow	-2	
crCross	-3	
crIBeam	-4	
crSize	-5	obsolete
crSizeNESW	-6	
crSizeNS	-7	
crSizeNWSE	-8	
crSizeWE	-9	
crUpArrow	-10	
crHourGlass	-11	
crDrag	-12	

图 5-4　系统内建光标及其常量值

具体到“BCB 画板”程序，首先我们在 Unit1.h 主窗体头文件中定义光标常量，注意不应和系统光标冲突。

```
const int crFill  = 5;
const int crPlus  = 6;
const int crDraw  = 7;
const int crErase = 8;
const int crText = 9;
const int crZoom = 10;
```

其次在主窗体的 OnCreate 相应函数中取得光标句柄并设置光标，在本例中是从文件中读入光标的。

```
Screen->Cursors[crFill] = LoadCursorFromFile("FILL.cur");
Screen->Cursors[crPlus] = LoadCursorFromFile("PLUS.cur");
Screen->Cursors[crDraw] = LoadCursorFromFile("DRAW.cur");
Screen->Cursors[crErase] = LoadCursorFromFile("ERASE.cur");
Screen->Cursors[crText] = LoadCursorFromFile("TEXT.cur");
```

```
Screen->Cursors[crZoom] = LoadCursorFromFile("ZOOM.cur");
```

此后在程序执行中就可以利用这些光标了。譬如，某一时刻希望鼠标移动到图画上时变成铅笔形状，可以使用下面以行代码：

```
Image->Cursor=TCursor(crDraw);
```

5.3.3 工具箱和颜料盒的实现

1. 工具箱响应规则

在主窗体头文件中，定义一个 TDrawingTool 枚举类型，其中每一值对应一个工具箱按钮：

```
enum TDrawingTool {dtSelect, dtPencil, dtZoomOut, dtZoomIn, dtBrush, dtFill, dtErase, dtText, dtFog,
dtLine ,dtRectangle , dtEllipse, dtRoundRect, dtSolidRect, dtSolidEllipse, dtSolidRndRect};
```

声明一个 TDrawingTool 类型的变量 ToolState，用于描述当前工具箱工具选择状态。

```
TDrawingTool ToolState;
```

如前文所述，工具箱的按钮响应函数并不完成具体的操作，而只是改变 ToolState 的值，设定当前 Image 的光标，以及一些与具体操作相关的初始化工作。现举几例。

```
void __fastcall TMainForm::SpeedButton10Click(TObject *Sender)
{     // 画直线
      EraseSelect();
      ToolState=dtLine;
      Image->Cursor=TCursor(crDraw);
}
void __fastcall TMainForm::SpeedButton11Click(TObject *Sender)
{     // 绘制文字
      EraseSelect();
      ToolState=dtText;
      // 将画刷设置为 bsClear，绘制“透明”文本
      Image->Canvas->Brush->Style=bsClear;
      Image->Cursor=TCursor(crText);
}
void __fastcall TMainForm::SpeedButton4Click(TObject *Sender)
{     // 区域选择
      ToolState=dtSelect;
      Image->Cursor=TCursor(crPlus);
}
void __fastcall TMainForm::SpeedButton1Click(TObject *Sender)
{     // 铅笔
      EraseSelect();
```

```
        ToolState=dtPencil;
        Image->Cursor=TCursor(crDraw);
    }
```

这里用到了我们定义的 EraseSelect 函数，此函数用于擦除选择区域并恢复画图状态。此函数在后文中具体分析。

具体绘图操作在 Image 对象的 OnMouseMove、OnMouseDown 和 OnMouseUp 事件的响应函数中，使用开关语句 switch……case……对每种工具箱状态分别实现。

2. 实现颜料盒

颜料盒用于选择前景色和背景色，注意在用户选择时，应处理颜色块 FGShape 和 BGShape，使得它们随时与 CColorGrid 组件中的选择一致。

响应 CColorGrid 组件的 OnChange 事件代码如下：

```
void __fastcall TMainForm::ColorGrid1Change(TObject *Sender)
{
    FGShape->Brush->Color = ColorGrid1->ForegroundColor;
    if (FGShape->Brush->Color == TColor(clBlack))
        FGShape->Pen->Color = TColor(clWhite);
    else
        FGShape->Pen->Color = TColor(clBlack);
    BGShape->Brush->Color = ColorGrid1->BackgroundColor;
    if (BGShape->Brush->Color == TColor(clBlack))
        BGShape->Pen->Color = TColor(clWhite);
    else
        BGShape->Pen->Color = TColor(clBlack);
}
```

请注意对 Shape 组件的处理。对于 Shape 组件来说，它的 Pen 属性的 Color 属性代表了画出形状的边框，而 Brush 属性的 Color 属性决定画出形状的填充颜色。如果用户选择了黑色，则用白色画边框，其他颜色则统一使用黑色画边框。

5.4 图像绘制——画图功能的实现

所有画图功能的实现都是使用 TCanvas 类提供的绘图方法，绘制图像是不难的，这里关键是要实现用鼠标进行绘图。特别是绘制规则图形（直线、矩形等）时的“橡皮筋”功能的实现，包含了较多技巧。

因为本程序中是在一个 Image 组件的 Canvas 属性上画图，所以不必考虑重画问题。

前文已经提到，所有的绘图代码都在 Image 对象的 OnMouseMove，OnMouseDown，OnMouseUp 事件的响应函数中。在实现绘图中，三个鼠标响应函数的利用是很关键的。鼠标响应函数传递 5 个重要的参数，如表 5-3 所列。

表 5-3　鼠标响应函数分析

参数	说　明
Sender	鼠标事件发生对象
Button	鼠标按键情况，有三种取值：mbLeft、mbRight、mbMiddle。OnMouseMove 事件响应函数不具有此参数
Shift	TShiftState 集合类型。描述鼠标动作时，Alt,Ctrl,Shift 按钮的状态。特别的，对于 OnMouseMove 事件，此参数还可以获知是否有鼠标键被按下
X，Y	事件发生时鼠标的坐标

下面分绘图方式的不同分别叙述其实现方法。

5.4.1　铅笔、画刷和橡皮

铅笔、画刷和橡皮其实都是使用 Canvas 的画笔画线的操作。铅笔是笔宽为 1 的画笔，画刷的笔宽由颜料盒上的 TCSpinEdit 组件设定。橡皮是采用背景色作画的画笔，在本程序中设定笔宽为 13。铅笔和画刷效果如图 5-5 所示。

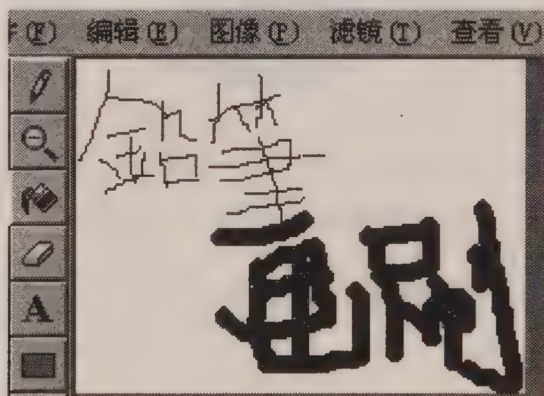


图 5-5　铅笔和画刷效果

铅笔、画刷和橡皮的绘制方式是用户按下鼠标左键不放，并移动鼠标，在图画上画出鼠标移过的轨迹。请看这三个工具是如何实现的。

在 ImageMouseDown 函数中：

```
// 判断按下的是否为左键，只有鼠标左键被按下才进行绘图操作。
if (Button != mbLeft)
      return;
switch (ToolState){
case dtPencil:
      Image->Canvas->Pen->Width=1;
      Image->Canvas->Pen->Color = FGShape->Brush->Color;
      Image->Canvas->Brush->Color = BGShape->Brush->Color;
      Image->Canvas->MoveTo(X,Y);
      break;
case dtBrush:
      Image->Canvas->Pen->Color = FGShape->Brush->Color;
```

```
        Image->Canvas->Brush->Color = BGShape->Brush->Color;
        Image->Canvas->MoveTo(X,Y);
        break;
    case dtErase:
        Image->Canvas->Pen->Color = BGShape->Brush->Color;
        Image->Canvas->Brush->Color = BGShape->Brush->Color;
        Image->Canvas->Pen->Width = 13;
        Image->Canvas->MoveTo(X,Y);
        break;
            ……
```

当鼠标左键按下时，进行必要的初始化工作，包括设定前景色（画笔颜色）和背景色（画刷颜色），设定笔宽，同时将画笔位置移动到鼠标位置。

在 ImageMouseMove 函数中：

```
TVarRec tempvar[2] = {X, Y};
StatusBar1->Panels->Items[1]->Text
= Format("当前位置: (%d, %d)", tempvar, 2);
if (!Shift.Contains(ssLeft))
    return;
switch (ToolState){
case dtPencil:
    Image->Canvas->LineTo(X,Y);
    break;
case dtBrush:
    Image->Canvas->LineTo(X,Y);
    break;
case dtErase:
    Image->Canvas->LineTo(X,Y);
    break;
```

函数首先在状态栏中显示当前所在点。这里使用了 Format 函数，此函数是一个非常有用的字符串处理函数，它的声明如下：

```
AnsiString __fastcall Format(const AnsiString Format, const System::TVarRec* Args, const int Args_Size);
```

此函数类似于标准输入输出函数中的 sprintf 函数。第一个参数是格式说明字符串，Args 参数指向一个共用体数组，Args_Size 参数说明 Args 中的变量或常量的个数。

通过 Shift 参数判断是否移动鼠标时鼠标左键被按下。如果是，将进行画图操作。画图使用了 Canvas 类的 LineTo 方法。

最后要在 ImageMouseUp 函数中做必要的善后工作。

```
if (Button != mbLeft)
    return;
switch (ToolState){
```

```
    case dtPencil:
        Image->Canvas->Pen->Width=PenWidth->Value;
        break;
    case dtErase:
        Image->Canvas->Pen->Width=PenWidth->Value;
        break;
```

5.4.2 颜料桶和喷枪

颜料桶和喷枪都是在 OnMouseDown 事件响应函数中直接完成的，不需要考虑处理 OnMouseMove 或 OnMouseUp 事件的问题。颜料桶和喷枪效果如图 5-6 所示。

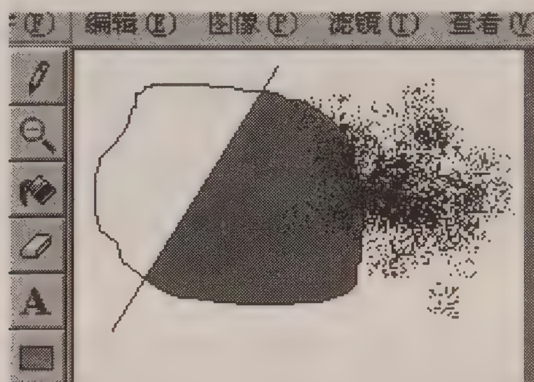


图 5-6　颜料桶和喷枪效果

颜料桶进行颜色填充操作，实现颜料桶需要使用 Canvas 类的 FloodFill 方法。在 ImageMouseDown 函数中加入：

```
if (ToolState==dtFill)
{
    if (Button == mbLeft)
        Image->Canvas->Brush->Color = FGShape->Brush->Color;
    else
        Image->Canvas->Brush->Color = BGShape->Brush->Color;
    Image->Canvas->FloodFill(X, Y, Image1->Canvas->Pixels[X][Y], fsSurface);
    return;
}
```

这段代码加在“if (Button != mbLeft) return;”一行之前，因为填充需要处理鼠标右键，右键用于使用背景色填充。

FloodFill 方法采用 fsSurface 模式进行填充，即当遇到与（X，Y）点颜色不同的点填充停止。

喷枪操作的实质是在一个方形区域内随机画点，有两个参数——方形边长大小和随机点的数量与喷枪的效果密切相关。在头文件中定义：

```
const int FogSize=14;
```

在 OnCreate 事件响应函数中初始化随机数：

```
randomize();
```

然后在 ImageMouseDown 函数中实现喷枪操作：

```
  switch (ToolState){
    ......
  case dtFog:
    for (int i=0;i<FogSize*FogSize/8;i++){
      Image->Canvas->
        Pixels[X+random(FogSize)][Y+random(FogSize)]
        =FGShape->Brush->Color;
    }
    break;
    ......
```

这里通过 TCanvas 类的 Pixels 属性设置在一个方形区域内某点的颜色值，在 FogSize*FogSize 大小的区域内有约 1/8 的随机点被画上了前景色。

5.4.3 放大缩小图像、绘制文字

放大缩小图像和绘制文字功能也都是在图像组件的 OnMouseDown 事件响应函数中直接完成的。放大缩小图像是 TCanvas 类的 StretchDraw 方法的直接应用。与绘制文字相关的是 TCanvas 类的 Font 属性和 TextOut 方法。

在头文件中定义放缩图像倍数常量：

```
const double ZoomSize=1.25;
```

而后在 ImageMouseDown 函数中加入：

```
switch (ToolState){
......
case dtZoomOut:
    tmpBitmap = new Graphics::TBitmap();
    tmpBitmap->Width = Image->Picture->Width*ZoomSize;
    tmpBitmap->Height = Image->Picture->Height*ZoomSize;
    tmpBitmap->Canvas->StretchDraw
      (TRect(0,0,tmpBitmap->Width,tmpBitmap->Height),
    Image->Picture->Graphic);
    Image->Picture->Graphic = tmpBitmap;
    delete tmpBitmap;
    break;
case dtZoomIn:
    tmpBitmap = new Graphics::TBitmap();
    tmpBitmap->Width = Image->Picture->Width/ZoomSize;
    tmpBitmap->Height = Image->Picture->Height/ZoomSize;
```

```
        tmpBitmap->Canvas->StretchDraw
            (TRect(0,0,tmpBitmap->Width,tmpBitmap->Height),
            Image->Picture->Graphic);
        Image->Picture->Graphic = tmpBitmap;
        delete tmpBitmap;
        break;
```

实现图像的放缩有以下三步：

(1) 在内存中建立临时位图。

(2) 应用 TCanvas 类的 StretchDraw 方法将 Image 组件上图像放缩后画到临时位图上。

(3) 将临时位图赋给 Image->Picture->Graphic，即在 Image 组件上显示出来。

绘制文字操作需要新建一个对话框 FontForm，如图 5-7 所示。

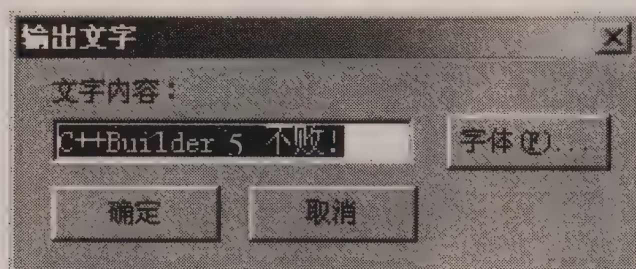


图 5-7 输出文字对话框

对话框中包含了字体对话框(FontDialog)组件，用于字体和字型的选择。因为对话框在程序中要使用 ShowModal 显示并获得返回值，所以设定“确定”和“取消”两个按键的 ModalResult 属性分别为 mrOK 和 mrCancel。

在 ImageMouseDown 函数中具体实现绘制文字的代码。

```
switch (ToolState){
......
case dtText:
    if (FontForm->ShowModal() == IDOK){
        Image->Canvas->Font
            =FontForm->FontDialog1->Font;
        Image->Canvas->TextOut(X,Y,FontForm->Edit1->Text);
    }
    break;
```

这里使用了 Canvas 对象的 TextOut 方法将文字按用户指定的字体字型画到鼠标单击处。Canvas 对象的 Font 属性指定输出文字的字体字型。

5.4.4 规则图形的绘制

在“BCB 画板”程序中，提供绘制的图形有：直线、实心矩形和矩形框，实心圆角矩形和圆角矩形框，椭圆面和椭圆。绘制这些图形利用了 TCanvas 类的绘图函数：LineTo（直线）、Rectangle（矩形）、Ellipse（椭圆）、RoundRect（圆角矩形）。

绘制规则图形时实现了一般绘图程序中流行的“橡皮筋”功能，即当用户开始画图操作时，随着鼠标的移动，用户能够即时的看到绘制图形的变化，就好像一个“橡皮筋”在不断改变形状一样。可以想象，应该响应 OnMouseMove 事件，把鼠标经过各点可能出现的形状都绘制出来。但另一个问题是，我们希望得到的是鼠标按钮按下和松开这两点所形成的图形，所以必须将鼠标上一次触发 OnMouseMove 事件绘制的图形擦去。

因为需要将上一次绘制的图形，所以必须纪录上一次鼠标的位置，同时也必须纪录鼠标的起始点以便最终绘出图形。这样需要分别定义变量：MovePt 和 Origin。

所有类型图形的实际绘图操作都在 DrawShape 函数中实现。图 5-8 给出了规则图形绘制的效果。

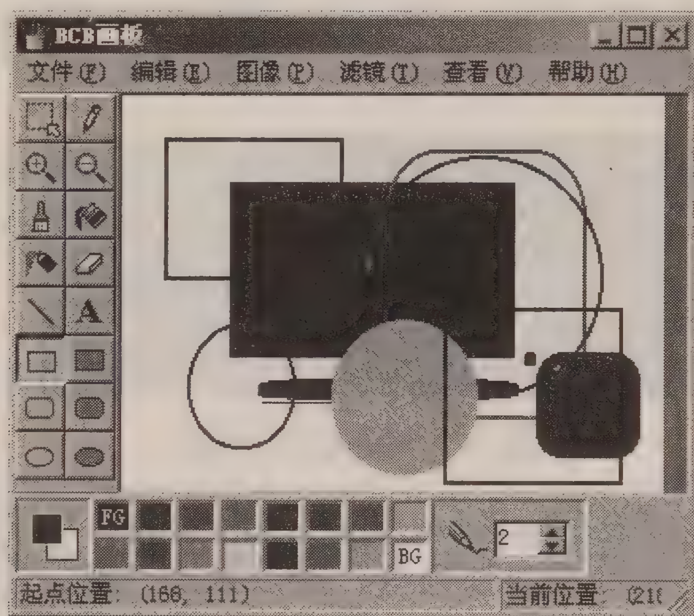


图 5-8　规则图形的绘制

Source 各种规则图形的绘制实现

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::DrawShape(TPoint TopLeft, TPoint BottomRight, TPenMode AMode)
{
    mage->Canvas->Pen->Mode = AMode;
    switch (ToolState){
    case dtLine:
        Image->Canvas->MoveTo(TopLeft.x, TopLeft.y);
        Image->Canvas->LineTo(BottomRight.x, BottomRight.y);
        break;
    case dtRectangle:
        Image->Canvas->Rectangle(TopLeft.x, TopLeft.y,BottomRight.x, BottomRight.y);
        break;
    case dtEllipse:
        Image->Canvas->Ellipse(TopLeft.x, TopLeft.y,BottomRight.x, BottomRight.y);
        break;
```

```
        case dtRoundRect:
            Image->Canvas->RoundRect(TopLeft.x, TopLeft.y, BottomRight.x,
                BottomRight.y, (TopLeft.x - BottomRight.x)/2, (TopLeft.y - BottomRight.y)/2);
            break;
        case dtSolidRect:
            Image->Canvas->Rectangle(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y);
            break;
        case dtSolidEllipse:
            Image->Canvas->Ellipse(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y);
            break;
        case dtSolidRndRect:
            Image->Canvas->RoundRect(TopLeft.x, TopLeft.y, BottomRight.x,
                BottomRight.y, (TopLeft.x - BottomRight.x)/2,
                (TopLeft.y - BottomRight.y)/2);
            break;
        case dtSelect:
            Image->Canvas->Rectangle(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y);
            break;
        }
        Image->Canvas->Pen->Mode = pmCopy;
    }
    //---------------------------------------------------------------------------
```

实现绘制实心图形和空心图形需要利用画刷和画笔。在绘制空心图形时，Pen 的 Color 属性应为前景色，Brush 的 Style 属性设为 bsClear，即不对图形填充。绘制实心图形，只需将 Pen 的 Color 属性和 Brush 的 Color 属性都设为前景色即可。这些工作应在 OnMouseDown 事件的响应函数中完成，同时，OnMouseDown 响应函数中还应纪录绘制的起点位置。

```
    switch (ToolState) {
    ……
    case dtSolidRect:
    case dtSolidEllipse:
    case dtSolidRndRect:
        Image->Canvas->Brush->Color = FGShape->Brush->Color;
        Image->Canvas->Pen->Color = FGShape->Brush->Color;
            Origin = Point(X, Y);
        StatusBar1->Panels->Items[0]->Text =
            Format("起点位置: (%d, %d)", tempvar, 2);
        MovePt = Origin;
        break;
    case dtLine:
    case dtRectangle:
```

```
case dtEllipse:
case dtRoundRect:
    Image->Canvas->Brush->Style=bsClear;
    Image->Canvas->Pen->Color = FGShape->Brush->Color;
    Origin = Point(X, Y);
    StatusBar1->Panels->Items[0]->Text =
        Format("起点位置: (%d, %d)", tempvar, 2);
    MovePt = Origin;
    break;
```

前面提到，在 OnMouseMove 事件的响应函数中应该随时画出图形，同时将上一次绘制的图形擦去。实现这一点用到了 TPen 类的 Mode 属性，此属性定义画图模式，正常情况下为 pmCopy，此处使用 pmNotXor 模式进行绘图。该模式的含义是：将笔的颜色和绘制点的颜色进行异或（Xor）组合并将颜色反转，为最终绘制出的颜色。使用这种模式时，进行完全相同的两次绘图操作后，绘制处的图形颜色将重新恢复为绘制前的颜色。具体请看下面的代码。

ImageMouseMove 函数中：

```
switch (ToolState){
……
case dtLine:
case dtRectangle:
case dtEllipse:
case dtRoundRect:
case dtSolidRect:
case dtSolidEllipse:
case dtSolidRndRect:
    DrawShape(Origin, MovePt,pmNotXor);
    MovePt = Point(X, Y);
    DrawShape(Origin, MovePt,pmNotXor);
    break;
```

在 OnMouseUp 事件的响应函数中最终完成绘图操作（使用 pmCopy 模式），并恢复画笔画刷的状态。如下：

```
switch (ToolState){
……
case dtLine:
case dtRectangle:
case dtEllipse:
case dtRoundRect:
case dtSolidRect:
case dtSolidEllipse:
case dtSolidRndRect:
```

```
    DrawShape(Origin, Point(X, Y), pmCopy);
    Image->Canvas->Brush->Style=bsSolid;
    Image->Canvas->Brush->Style=
        BGShape->Brush->Color;
    break;
```

这样具有专业风格的画图操作就实现了。

5.5 区域选择和图像的剪贴、复制

在一个成熟的图像编辑处理程序中，图像区域的剪贴、复制功能必不可少。显然这些功能应通过 Windows 剪贴板来实现。

然而这同时带来了另一个问题，即如何实现在图像上进行区域选择，以确定剪贴或者复制的部分。直观的区域选择功能一般是在选中的部分画上虚线框，实现这一目的必须使用 Pen 的 Style 属性。

5.5.1 区域选择的实现

选择区域有点像前面所述的绘制规则图形。在用户选择区域时，其实是一个虚线的矩形框在不断的被绘制、擦除。所以在 OnMouseDown 和 OnMouseMove 的处理部分，与绘制规则图形是基本相同的。但在 OnMouseUp 事件的响应函数中，区域选择框不能像绘制矩形那样被“确认”，因为选择框最终还是要被擦除的。

绘制虚线框应该设置 Pen 的 Style 属性为 psDot，同时笔宽应设为 1，否则 psDot 的 Style 会无效。

以下是鼠标响应函数中的相应代码。

ImageMouseDown 函数中：

```
case dtSelect:
    EraseSelect();
    Image->Canvas->Brush->Style=bsClear;
    Image->Canvas->Pen->Style = psDot;
    Image->Canvas->Pen->Width = 1;
    Origin = Point(X, Y);
    MovePt = Origin;
    break;
```

ImageMouseMove 函数中：

```
case dtSelect:
    DrawShape(Origin, MovePt,pmNotXor);
    MovePt = Point(X, Y);
    DrawShape(Origin, MovePt,pmNotXor);
```

```
        break;
```

ImageMouseUp 函数中：

```
case dtSelect:
        Image->Canvas->Brush->Style=bsSolid;
        Image->Canvas->Pen->Width=PenWidth->Value;
        Image->Canvas->Pen->Style = psSolid;
        MovePt=Point(X,Y);
        isSelect=true;   //标识已有区域被选择
        SelectRect=(TRect(Origin, MovePt));   //选择的区域
        T1->Enabled=true;    //“剪切”菜单项有效
        C1->Enabled=true;    //“拷贝”菜单项有效
        break;
```

当一个区域已被选择，用户又放弃这个选择时，应将之前绘制的选择框擦去，并进行放弃选择的相关处理。这些工作由 EraseSelect 函数完成。

```
void TMainForm::EraseSelect()
{
        if (isSelect){
                //设置画笔画刷，用于擦去选择框
                Image->Canvas->Brush->Style=bsClear;
                Image->Canvas->Pen->Style = psDot;
                Image->Canvas->Pen->Width = 1;
                isSelect=false;
                DrawShape(Origin, MovePt,pmNotXor);
                T1->Enabled=false;   //“剪切”菜单项失效
                C1->Enabled=false;   // “拷贝”菜单项失效
                //恢复画笔画刷状态
                Image->Canvas->Brush->Style=bsSolid;
                Image->Canvas->Pen->Style = psSolid;
                Image->Canvas->Pen->Width=PenWidth->Value;
        }
}
```

5.5.2 应用剪贴板

当用户在图像上选好一个区域后，就可以进行剪切和复制操作了。另一方面，当剪贴板上有图像时，用户可以选择“粘贴”将剪贴板上的图像复制到编辑的图像上。实现这些功能涉及到剪贴板类 TClipboard 相关内容。

本质上，剪贴板只是一个全局内存块。当一个应用程序将数据传送给剪贴板后，通过修改内存块分配标志，把相关内存块的所有权从应用程序移交给 Windows 自身。其他应用程序

可以通过一个句柄找到这个内存块，从而能够从内存块中读取数据，这样就实现了数据在不同应用程序间的传输。

在 C++ Builder 中使用 Clipboard 函数获得全局剪贴板对象，以实现对 Windows 剪贴板的操作。

利用剪贴板类可以进行文本、图像和部件的传输，剪贴板类为实现这些方法提供了相应的属性和方法。TClipboard 的属性有：AsText，放置或获取剪贴板的文本；FormatCount，获取可用剪贴板格式的数目；Formats，可用剪贴板格式链。

TClipboard 类的重要方法有：

- Clear 方法。

清除剪贴板的内容。

- Assign 方法。

声明：void __fastcall Assign(Classes::TPersistent* Source);

把指定的对象拷贝到剪贴板，用于图形、图像对象。

- Open 方法。

打开剪贴板，阻止其他应用程序改变它的内容。

- Close 方法。

关闭打开的剪贴板。

- SetAsHandle 方法。

声明：void __fastcall SetAsHandle(Word Format, int Value);

把指定格式数据的句柄交给剪贴板。

- GetAsHandle 方法。

声明：int __fastcall GetAsHandle(Word Format);

返回剪贴板指定格式数据的句柄。

- HasFormat 方法。

判断剪贴板是否支持指定的格式。

- SetTextBuf 方法。

声明：void __fastcall SetTextBuf(char * Buffer);

设置剪贴板的文本内容。

具体在“BCB 画板”程序中，实现图像拷贝、剪切、粘贴的代码如下。

Source 实现图像拷贝、剪切、粘贴的代码

```
void __fastcall TMainForm::C1Click(TObject *Sender)
{
    // 图像复制
    EraseSelect();
    Graphics::TBitmap *tmpBitmap;
    tmpBitmap = new Graphics::TBitmap();
    tmpBitmap->Width=SelectRect.Right-SelectRect.Left;
    tmpBitmap->Height=SelectRect.Bottom-SelectRect.Top;
```

```
        tmpBitmap->Canvas->CopyRect(
        TRect(0,0,tmpBitmap->Width,tmpBitmap->Height),
        Image->Canvas,SelectRect);
        Clipboard()->Assign(tmpBitmap);
        delete tmpBitmap;
        T1->Enabled=false;
        C1->Enabled=false;
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::T1Click(TObject *Sender)
{
        // 图像剪切
        EraseSelect();
        C1Click(Sender);
        Image->Canvas->CopyMode = cmWhiteness;
        Image->Canvas->CopyRect(SelectRect, Image->Canvas, SelectRect);
        Image->Canvas->CopyMode = cmSrcCopy;
        T1->Enabled=false;
        C1->Enabled=false;
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::N3Click(TObject *Sender)
{
        // 图像粘贴
        Graphics::TBitmap *tmpBitmap;
        if (Clipboard()->HasFormat(CF_BITMAP)){
                tmpBitmap = new Graphics::TBitmap();
                try{
                        tmpBitmap->Assign(Clipboard());
                        Image->Canvas->Draw(0, 0, tmpBitmap);
                        delete tmpBitmap;
                }
                catch(...){
                        delete tmpBitmap;
                }
        }
}
//---------------------------------------------------------------------------
```

复制时首先建立一个临时位图，将用户选择区域内的图像拷贝到临时位图（使用 TCanvas 类的 CopyRect 方法），并利用剪贴板类的 Assign 方法将临时位图的内容送到剪贴

板。

图像剪切分为两步：

(1) 复制选择区域图像。

(2) 将选择区域图像擦除。在程序中将选择区域填充为白色。

执行粘贴前先用 HasFormat 方法判断剪贴板中是否含有位图，CF_BITMAP 是位图格式常量。表 5-4 为剪贴板中可能的数据格式常量及含义。

表 5-4 剪贴板数据格式常量及含义

数据格式	含　义
CF_BITMAP	Windows 位图
CF_METAFILE	图元文件
CF_PICTURE	TPicture 类型的对象
CF_TEXT	文本。每行以 CF_LF 结束，空字符'\0'标志文本结束
CF_OBJECT	任何 TPersistent 类型的对象，譬如可以是 TJPEGImage 类的图像

如果检查到存在位图，则使用 TBitmap 的 Assign 方法将剪贴板内容复制到一个临时位图上，最后用 Draw 方法将临时位图画在编辑的图像上。

上述五种格式是 C++ Builder 或 VCL 组件专用的格式。另外，Windows API 还定义了很多剪贴板格式，包括：

CF_BITMAP	CF_DSPMETAFILEPICT
CF_OWNERDISPLAY	CF_SYLK
CF_DIB	CF_DSPTEXT
CF_PALETTE	CF_TEXT
CF_DIF	CF_METAFILEPICT
CF_PENDATA	CF_TIFF
CF_DSPBITMAP	CF_OEMTEXT
CF_RIFF	CF_WAVE

在 C++ Builder 中也可以使用这些剪贴板数据格式。尽管 C++ Builder 没有提供特定的支持来检索这些数据类型，但 TClipboard 类可以正确地从 Windows 剪贴板获取并处理不同类型的数据。

5.6 新建、打开、存储文件及简单图像处理

5.6.1 新建、打开、存储文件

打开一个图像文件进行编辑可以直接使用 Image 组件的 Picture 属性的 LoadFormFile 方法，但注意应该对 JPEG 格式的图像进行特殊处理，因为 TJPEGImage 类不具有 Canvas 属性，使用 LoadFormFile 方法装入 Image 对象后，Image 对象的 Canvas 属性也不能进入。所以，对于

JPEG 格式图像应该做必要的转换。

```
void __fastcall TMainForm::O1Click(TObject *Sender)
{
        // 打开图像文件
        if (OpenDialog1->Execute()){
                CurrentFile = OpenDialog1->FileName;
                if (ExtractFileExt(OpenDialog1->FileName)==".jpg"){
                        TJPEGImage *JpegImage1=new TJPEGImage();
                        Graphics::TBitmap *Bit=new Graphics::TBitmap();
                        JpegImage1->LoadFromFile(CurrentFile);
                        Bit->Width = JpegImage1->Width ;
                        // 使位图与 JPEG 图像尺寸相等
                        Bit->Height = JpegImage1->Height ;
                        Bit->Canvas->StretchDraw(Rect(0,0,Bit->Width,Bit->Height),JpegImage1);
                        Image->Picture->Graphic=Bit;
                        delete JpegImage1;
                        delete Bit;
                }
                else
                        Image->Picture->LoadFromFile(CurrentFile);
        }
}
```

在“BCB 画板”的保存和另存为功能中并未加入对 JPEG 格式的支持，但利用上一章讲述的图像转换技术，实现这一点并不困难，将 BCBSee32 中相关代码稍做改动即可。

```
void __fastcall TMainForm::S1Click(TObject *Sender)
{       // 保存图像文件
        if (CurrentFile != EmptyStr&&ExtractFileExt(CurrentFile)!=".jpg"){
                Image->Picture->SaveToFile(CurrentFile);
        }
        else{
                A1Click(Sender); // SaveAs
        }
}
void __fastcall TMainForm::A1Click(TObject *Sender)
{       // 另存为
        if (SaveDialog1->Execute()){
                CurrentFile = SaveDialog1->FileName;
                S1Click(Sender); // Save
        }
}
```

实现新建图像文件必须再创建一个对话框，用于输入新建图像的尺寸。

```
void __fastcall TMainForm::N1Click(TObject *Sender)
{     // 新建图像
      NewBMPForm->WidthEdit->Text =
            IntToStr(Image->Picture->Width);
      NewBMPForm->HeightEdit->Text =
            IntToStr(Image->Picture->Height);
      if (NewBMPForm->ShowModal() != IDCANCEL){
            Graphics::TBitmap *Bitmap = new Graphics::TBitmap();
            Bitmap->Width = StrToInt(NewBMPForm->WidthEdit->Text);
            Bitmap->Height = StrToInt(NewBMPForm->HeightEdit->Text);
            Image->Picture->Graphic = Bitmap;
            CurrentFile = EmptyStr;
      }
}
```

图 5-9 给出了新建位图文件对话框。

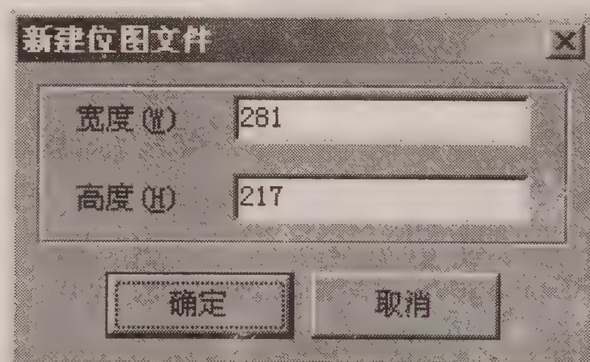


图 5-9　新建位图文件对话框

5.6.2　尺寸设置、反色及图像打印

图像尺寸设置原理与放大缩小图像相同，分以下三步实现：

（1）建立临时位图。

（2）使用 TCanvas 类的 StretchDraw 方法将 Image 组件上的图像放缩后画在临时位图上。

（3）将临时位图赋给 Image 组件的 Picture->Graphic。

首先创建一个对话框，用于输入新的图像尺寸。

代码部分如下：

```
void __fastcall TMainForm::S2Click(TObject *Sender)
{
      ResizeBMPForm->WidthEdit->Text=IntToStr(Image->Width);
      ResizeBMPForm->HeightEdit->Text=IntToStr(Image->Height);
      if(ResizeBMPForm->ShowModal()==IDOK){
```

```
            Graphics::TBitmap *tmpBitmap =
                new Graphics::TBitmap();
            tmpBitmap->Width =
                StrToInt(ResizeBMPForm->WidthEdit->Text);
            tmpBitmap->Height =
                StrToInt(ResizeBMPForm->HeightEdit->Text);
            tmpBitmap->Canvas->StretchDraw
                 (TRect(0,0,tmpBitmap->Width,tmpBitmap->Height),
            Image->Picture->Graphic);
            Image->Picture->Graphic=tmpBitmap;
            delete tmpBitmap;
        }
    }
```

Windows 画图程序有反色的功能。反色又称为反像或负像，它是用最大值 255 分别减去图像中各个位置的红绿蓝三基色的值，得到新的三基色值。这样得到的图像具有底片效果。

按上述思路实现图像反色是很自然的，需要用到 TCanvas 类的 Pixels 属性。

```
void __fastcall TMainForm::N6Click(TObject *Sender)
{
    DWORD Color;
    for (int i=0;i<Image->Height;i++){
        for (int j=0;j<Image->Width;j++){
            Color=DWORD(Image->Canvas->Pixels[j][i]);
            Image->Canvas->Pixels[j][i]=RGB(
                0xFF-BYTE(Color),
                0xFF-BYTE(Color>>8),
                0xFF-BYTE(Color>>16));
        }
    }
}
```

利用位运算右移“>>”分离出 RGB 三基色值，分别用最大值 0xFF（255）剪去三基色的值，使用 RGB()宏将新三基色值再赋给图像上的点。

RGB 宏将 RGB 三个 BYTE 值组合成一个 DWORD 值，（TColor 其实也是一个 DWORD）定义如下：

```
#define RGB(r, g ,b)  ((DWORD) (((BYTE) (r) | ((WORD) (g) << 8)) |
    (((DWORD) (BYTE) (b)) << 16)))
```

上述实现方法好像很自然，但其实它是笔者所能想到的最慢的方法。当图像稍大时，如一个 800*600 的图像，使用这种方法进行反色处理要用几十秒，这显然是不能接受的。我们将在后文中讨论并解决这个问题。

图像打印功能在第 4 章的 BCBSee32 程序中已经讲解，可以不加改动的将 BCBSee32 程序中的图像打印代码套用过来。

5.7 图像处理高级话题

在上一节中，我们实现了“反色”这一简单的图像处理功能，所用的方法是直接改变 Image 组件画布上像素点的颜色。虽然这种方法是完全可行的，但它的速度实在让人不敢恭维。

另一方面，在 Windows 的画图程序中，反色操作是很快的。Photoshop 中的更加复杂的图像处理操作也都有很高的速度，这一切意味着存在一种更好的图像处理技术。

5.7.1 提升速度

上一节中实现反色操作的代码简单的从两个方向上扫描图像，并通过 TCanvas 类的 Pixels 属性访问图像中的每个像素。在那个循环体中做的另一件事是不断通过 TCanvas 类的 Pixels 属性设置每个像素点的颜色，这正是速度低下的罪魁祸首。因为，应用程序每一次调用 Pixels 属性设置颜色，都可能会引起图像的重画，同时这种调用也在不断的和显示内存交换数据，这显然导致了运行速度的大大降低。

在内存中操作图像是解决这个问题的有效办法。其具体方法是，在内存中建立一个临时位图，将 Image 组件的图像信息复制到临时位图中，然后对内存位图进行同样的反色处理，待处理完成后，把内存位图赋给 Graphic 属性，一次将结果显示出来。

这种方法避免频繁地对显示内存操作，从而使速度大大提升。其速度约为直接处理的 4 到 5 倍。

另一项改善图像处理速度的技术是使用 TBitmap 类的 ScanLine 属性。这种方法同样需要在内存中建立临时位图，但利用了 ScanLine 属性而非 Pixels 属性访问数据。该属性可以返回指向实际位图内存的指针，并通过该指针一次访问和处理位图的一整行数据，从而大大加快程序的运行速度。但是应该注意，使用 ScanLine 属性必须明确位图的内部存储形式：在 24 位真彩位图的情况下，每点的颜色信息由蓝、绿、红顺序（而非 TColor 中的 RGB 顺序）的三个字节组成。使用 ScanLine 属性访问位图非常之快，它的处理速度约是直接处理的 90 到 100 倍！

为了检验上述结果，并具体实践上述技术，下面给出一个分别使用上述三种方法对图像进行灰阶处理的范例程序。灰阶处理是将一个彩色图形转化为灰阶图像，由于“灰色”是所有色彩中 R\G\B 三基色值相等的色彩，三基色的值也就是色彩的亮度，所以转换时只需计算像素点的亮度即可。应用以下公式：

Gray = 0.299 * R +0.587 * G +0.114 * B

此公式是根据人眼对不同颜色的相对敏感度得来的。

直接处理方法的代码为：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    BYTE *PColor;
```

```
        DWORD Color;
        long Gray;
        TimeStart();
        for (int i=0;i<Image->Height;i++)
                for (int j=0;j<Image->Width;j++){
                        Color=DWORD(Image->Canvas->Pixels[j][i]);
                        PColor=(BYTE *)&Color;
                        Gray=299*PColor[0]+587*PColor[1]+114*PColor[2];
                        Image->Canvas->Pixels[j][i]=
                                RGB(Gray/1000,Gray/1000,Gray/1000);
        }
        TimeEnd();
}
```

使用内存位图处理的代码为：

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        BYTE *PColor;
        DWORD Color;
        long Gray;
        Graphics::TBitmap *tmpBitmap=new Graphics::TBitmap();
        tmpBitmap->Assign((TPersistent*)Image->Picture->Graphic);
        TimeStart();
        for (int i=0;i<tmpBitmap->Height;i++)
                for (int j=0;j<tmpBitmap->Width;j++){
                        Color=DWORD(tmpBitmap->Canvas->Pixels[j][i]);
                        PColor=(BYTE *)&Color;
                        Gray=299*PColor[0]+587*PColor[1]+114*PColor[2];
                        tmpBitmap->Canvas->Pixels[j][i]=
                                RGB(Gray/1000,Gray/1000,Gray/1000);
                }
        Image->Picture->Graphic=tmpBitmap;
        TimeEnd();
        delete tmpBitmap;
}
```

使用 ScanLine 技术处理的代码为：

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
        long Gray;
        BYTE *ptr;
        Graphics::TBitmap *tmpBitmap=new Graphics::TBitmap();
```

```
        tmpBitmap->Assign((TPersistent*)Image->Picture->Graphic);
        tmpBitmap->PixelFormat=pf24bit;
        TimeStart();
        for (int y = 0; y < tmpBitmap->Height; y++)
        {
            ptr =(BYTE *) tmpBitmap->ScanLine[y];
            for (int x = 0; x < tmpBitmap->Width*3; x+=3){
                Gray=299*ptr[x+2]+587*ptr[x+1]+114*ptr[x];
                ptr[x]=Gray/1000;
                ptr[x+1]=Gray/1000;
                ptr[x+2]=Gray/1000;
            }
        }
        Image->Picture->Graphic=tmpBitmap;
        TimeEnd();
        delete tmpBitmap;
    }
```

以上三段代码包含在“图像处理算法比较”的范例程序中。此程序中编制了 TimeState 函数和 TimeEnd 函数用于实现计时工作。注意在使用 ScanLine 技术处理时，必须先强制设置 tmpBitmap 的 PixelFormat 属性为 pf24bit，否则如果图像的像素颜色深度不是 24 位，图像存储方式就不是每像素三字节，这将会造成严重的错误。

运行这个程序，将 TImage 组件中装入的 500×375 大小的图像转换为灰阶，直接处理用时 4.89s，内存位图处理用时 1.27s，应用 ScanLine 技术处理用时 0.05s。以上结果是在笔者的 Celeron 300A（超 450）计算机上得到的。结果如图 5-10 所示。



图 5-10　三种图像处理算法的速度比较

5.7.2 图像色彩调整

本节将讨论几种常用的图像色彩调整技术，包括调整对比度、调整亮度和色彩平衡。这些功能以及上节实现的灰阶处理都包含在“BCB 画板”程序中，实现这些图像处理功能采用了最快的 ScanLine 技术。

1. 调整对比度

图像对比度的增加使图像显得更鲜明。调整对比度对红、绿、蓝三基色的值使用下列公式调整：

Color = (Color – 128)* s +128

其中 Color 为基色的值，s 是调整系数，s>1 时增加对比度，s<1 是降低对比度。128 是三基色最大值 255 的中值，图像在对比度调整后平均亮度保持不变。

下面列出调整对比度函数 DoContrast 的代码。

Source 调整图像对比度

```
//---------------------------------------------------------------------------
void TMainForm::DoContrast(int s)
{
      int XX;
      BYTE *ptr;
      Graphics::TBitmap *tmpBitmap=new Graphics::TBitmap();
      tmpBitmap->Assign((TPersistent*)Image->Picture->Graphic);
      tmpBitmap->PixelFormat=pf24bit;
      for (int y = 0; y < tmpBitmap->Height; y++)
      {
            ptr =(BYTE *) tmpBitmap->ScanLine[y];
            for (int x = 0; x < tmpBitmap->Width*3; x+=3){
                  XX=((ptr[x+2]-128)*s+12800)/100;
                  XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+2]=XX;
                  XX=((ptr[x+1]-128)*s+12800)/100;
                  XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+1]=XX;
                  XX=((ptr[x]-128)*s+12800)/100;
                  XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+0]=XX;
            }
      }
      Image->Picture->Graphic=tmpBitmap;
      delete tmpBitmap;
```

```
}
//------------------------------------------------------------------------
```

特别注意的是在处理中应保证三基色的值小于 255 且大于 0，定义宏 MAX 和 MIN 控制颜色值在 0～255 之间：

```
#define MIN(A,B) (A<B)?A:B
#define MAX(A,B) (A>B)?A:B
```

在“BCB 画板”中分别使用 s=110 和 s=91（相当于原公式中的 1.1 和 0.91）来完成增加对比度和减小对比度的操作。

2. 调整亮度

调整亮度可以比较容易的通过编程来实现。简单的将红、绿、蓝三基色值分别增加或减小一个常数值，即可实现调整亮度的处理。

下面列出调整亮度函数 DoBright 的代码。

Source 调整图像亮度

```
//----------------------------------------------------------------------------
void TMainForm::DoBright(int b)
{
    int XX;
    BYTE *ptr;
    Graphics::TBitmap *tmpBitmap=new Graphics::TBitmap();
    tmpBitmap->Assign((TPersistent*)Image->Picture->Graphic);
    tmpBitmap->PixelFormat=pf24bit;
    for (int y = 0; y < tmpBitmap->Height; y++)
    {
        ptr =(BYTE *) tmpBitmap->ScanLine[y];
        for (int x = 0; x < tmpBitmap->Width*3; x+=3){
            XX=ptr[x+2]+b;
            XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+2]=XX;
            XX=ptr[x+1]+b;
            XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+1]=XX;
            XX=ptr[x]+b;
            XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+0]=XX;
        }
    }
    Image->Picture->Graphic=tmpBitmap;
    delete tmpBitmap;
}
//------------------------------------------------------------------------
```

在“BCB 画板”中分别使用 b=12 和 b=-12 来完成增加亮度和减小亮度的操作。

3. 色彩平衡

类似 PhotoShop 中的色彩平衡操作，色彩平衡允许用户随意增强或减弱图像中某一基色的强度。首先创建一个用于调整色彩的对话框，包含三个 TrackBar 组件，其范围是–100 ~ +100，如图 5-11 所示。

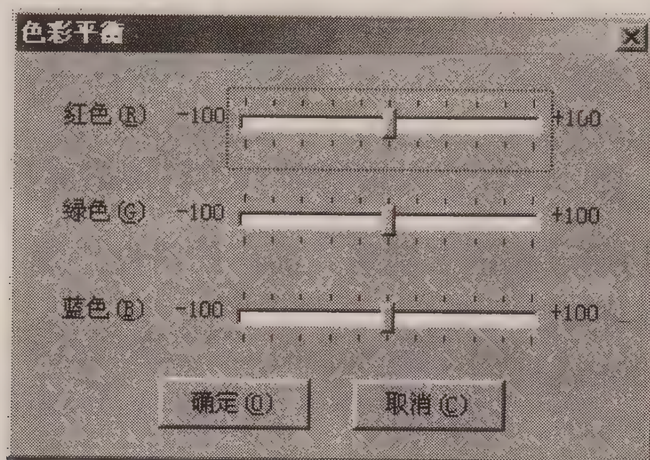


图 5-11　色彩平衡对话框

色彩平衡同样对基色值增加或减小一个常值。下面列出实现色彩平衡操作的代码：

```
void __fastcall TMainForm::N12Click(TObject *Sender)
{
    int XX;
    BYTE *ptr;
    Graphics::TBitmap *tmpBitmap=new Graphics::TBitmap();
    tmpBitmap->Assign((TPersistent*)Image->Picture->Graphic);
    tmpBitmap->PixelFormat=pf24bit;
    CForm->TrackBar1->Position=0;
    CForm->TrackBar2->Position=0;
    CForm->TrackBar3->Position=0;
    if (CForm->ShowModal()==IDOK){
        for (int y = 0; y < tmpBitmap->Height; y++){
            ptr =(BYTE *) tmpBitmap->ScanLine[y];
            for (int x = 0; x < tmpBitmap->Width*3; x+=3){
                XX=ptr[x+2]+CForm->TrackBar1->Position;
                XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+2]=XX;
                XX=ptr[x+1]+CForm->TrackBar2->Position;
                XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+1]=XX;
                XX=ptr[x]+CForm->TrackBar3->Position;
                XX=MAX(XX,0);XX=MIN(XX,255);ptr[x+0]=XX;
            }
        }
        Image->Picture->Graphic=tmpBitmap;
```

```
        }
        delete tmpBitmap;
    }
```

4. 滤镜技术

与色彩调整不同，对图像进行滤镜处理不是仅对一点的颜色值进行处理，而是将本点和其周围点的颜色值进行综合处理后得出本点的颜色值。滤镜技术最简单的算法称为“过滤盒”。它是通过建立一个方形的区域，对方形区域内的中心点颜色值和周围点颜色值进行加权和计算，得出中心点被处理颜色的值。在“BCB 画板”中实现了简单的滤镜处理，包括模糊、锐化、雕刻，这些处理都是通过建立一个 3*3 的过滤盒实现的。

利用过滤盒实现滤镜操作首先要建立一个过滤盒，即给定 3*3 过滤盒中各点的加权系数，然后按给定的系数处理过滤任务。DoFilter 函数实现了过滤的处理。

Source 实现通过过滤盒的图像滤波处理

```
//---------------------------------------------------------------------------
void TMainForm::DoFilter(int * flt, int Div)
{
        int XX[3];
        BYTE *ptr,*ptru,*ptrd,*ptr1;
        Graphics::TBitmap *Bitmap=new Graphics::TBitmap();
        Graphics::TBitmap *tmpBitmap=new Graphics::TBitmap();
        Bitmap->Assign((TPersistent*)Image->Picture->Graphic);
        Bitmap->PixelFormat=pf24bit;
        tmpBitmap->Assign((TPersistent*)Image->Picture->Graphic);
        tmpBitmap->PixelFormat=pf24bit;
        for (int y = 1; y < tmpBitmap->Height-1; y++)
        {
                ptr=(BYTE *)Bitmap->ScanLine[y];
                ptr1=(BYTE *)tmpBitmap->ScanLine[y];
                ptru=(BYTE *)tmpBitmap->ScanLine[y-1];
                ptrd=(BYTE *)tmpBitmap->ScanLine[y+1];
                for (int x=3;x<(tmpBitmap->Width-1)*3;x+=3){
                        XX[0]=0;XX[1]=0;XX[2]=0;
                        for (int i=-1;i<=1;i++)
                        for (int j=0;j<3;j++)
                                XX[j]+=ptr1[x+3*i+j]*flt[4+i];
                        for (int i=-1;i<=1;i++)
                        for (int j=0;j<3;j++)
                                XX[j]+=ptru[x+3*i+j]*flt[1+i];
```

```
                for (int i=-1;i<=1;i++)
                for (int j=0;j<3;j++)
                    XX[j]+=ptrd[x+3*i+j]*flt[7+i];
                for (int i=0;i<3;i++){
                    XX[i]=XX[i]/Div;
                    XX[i]=MAX(XX[i],0);
                    XX[i]=MIN(XX[i],255);
                    ptr[x+i]=XX[i];
                }
            }
        }
        Image->Picture->Graphic=Bitmap;
        delete tmpBitmap;
        delete Bitmap;
    }
    //---------------------------------------------------------------------------
```

图像模糊的原理是在图像中每个点的颜色中加入周围的颜色信息，从而减少中心颜色所占的比例。其实现的代码如下：

```
void __fastcall TMainForm::N15Click(TObject *Sender)
{
    int flt[9];
    flt[0]=5;flt[1]=5;flt[2]=5;
    flt[3]=5;flt[4]=60;flt[5]=5;
    flt[6]=5;flt[7]=5;flt[8]=5;
    DoFilter(flt,100);
}
```

图 5-12 给出了图像模糊处理后的效果。



图 5-12　几次图像模糊处理后获得的效果

图像锐化是突出原图像中每个颜色值，以降低其周围颜色的影响，增强图像细节。其实现代码如下：

```
void __fastcall TMainForm::N16Click(TObject *Sender)
{
    int flt[9];
    flt[0]=0;flt[1]=-5;flt[2]=0;
    flt[3]=-5;flt[4]=30;flt[5]=-5;
    flt[6]=0;flt[7]=-5;flt[8]=0;
    DoFilter(flt,10);
}
```

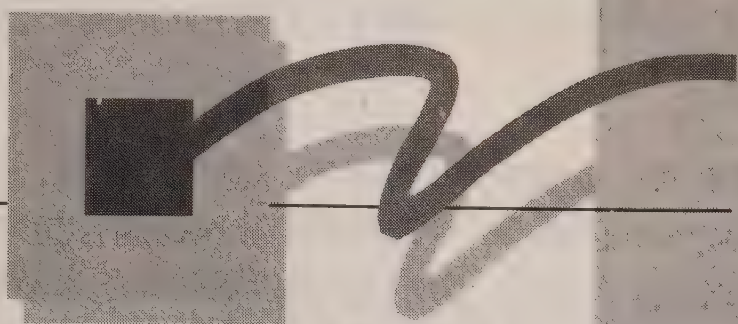
雕刻效果就像是图像被刻在金属板上一样。其实现代码如下：

```
void __fastcall TMainForm::N18Click(TObject *Sender)
{
    int flt[9];
    flt[0]=-15;flt[1]=-15;flt[2]=0;
    flt[3]=-15;flt[4]=15;flt[5]=15;
    flt[6]=0;flt[7]=15;flt[8]=0;
    DoFilter(flt,10);
}
```

第6章

多样 Windows 屏幕保护程序
——动画技术与图形技巧显示

动画实现技术：双缓冲区及掩图
Windows 屏保程序的实现
消息映射及处理
操作系统注册表
多媒体定时器及其他话题



本章导读

动画是一个颇有意思的话题，适当应用动画可以使应用程序情趣倍增。像电脑游戏这类程序，动画可能是整个程序的核心。

本章将完成 BCB 屏幕保护程序。编写一个真正完整的屏幕保护程序并非一件易事，它要考虑和处理的问题可能要比想象的多得多。但本章的任务并非是讲述如何做一个出色的屏保程序，其重点是讲解 Windows 动画技术和图形技巧显示的内容，所以 BCB 屏幕保护程序中将尽量省略一些繁琐的东西。然而即使这样，在本章的屏保范例中，仍然涵盖了较多的 Windows 编程技术，包括 Windows 消息的拦截和处理、利用注册表、应用程序参数传递等。这些内容同样是本章的重点。

本章内容包括:

- Windows 动画技术，包括实现动画中常用的双缓冲区和掩图技术。
- 具体实现范例程序“BCB 屏幕保护程序”，通过该程序的实现，讲述和演示动画生成和图像技巧显示。同时，也将讲述处理 Windows 消息、使用注册表等重要技术。
- 关于动画的其他话题。包括桌面精灵动画，多媒体定时器和三维动画技术的内容。

6.1　Windows 动画技术

一般来说，实现动画采用两种方法中的一种：基于精灵模式的动画和基于帧模式的动画。帧模式的动画是用一种连续的静止画面组合起来给人以连续运动的感觉，它实际上就是电影或卡通的工作方式。这种方式的动画实现起来没有特别的难度。

另一种方式的动画是基于精灵模式的动画，这种方式是真正的“生成动画”技术，它是在一个不变的背景上移动一个物体，例如游戏程序中用户控制的主角。基于精灵模式的动画具有更大的灵活性，并给了程序人员更大的发挥余地。

本章讨论的范围仅限于精灵模式的动画技术。

6.1.1　双缓冲区（Double Buffer）

开发人员创建动画并移动图形，重要的是让用户感到动画是平滑而又连续运动的。然而，如果仅仅简单的按照绘制一张图片—擦除这张图片—绘制下一张图片编制程序，这样创建的动画将出现强烈闪烁，无法让人接受。

解决这个问题可以采用双缓冲区的技术。双缓冲区（double buffer）是指绘制一组动画时，采用两个面，一个面被显示，而另一个面作为绘制面。当绘制面进行的动画处理结束后，缓冲区调换位置，快速显示被隐藏的绘制面，或者将被隐藏的缓冲区快速拷贝到显示缓冲区中。

在 C++ Builder 中编制动画，可以采用一种简单的双缓冲区技术，即在内存中建立一个临时位图作为后台缓冲区，在内存中处理图像，并在绘制画面时，使用 TCanvas 类的 CopyRect 方法将处理完成的画面拷贝到显示缓冲区中。

在内存中处理图像，这是所有动画方法依赖的基本原理。在内存中处理图像会提高处理速度，这是因为各种操作在系统内存中比在显示内存中执行起来快得多。同时因为中间画面不会再被显示在屏幕上，因而大大减小了屏幕闪烁。

借助 C++ Builder 的强大功能可以方便的生成精灵动画，利用在前面两章中所讲述的图形图像类，包括 TCanvas、TBitmap、TPaintBox 等等，将大大简化装载图像、在内存中处理图像和在屏幕上显示动画的操作。

6.1.2　TPaintBox 组件和 TTimer 组件

TPaintBox 组件和 TTimer 组件没有任何关系，但在动画编程中，它们都是必须的。

1. TPaintBox 组件

在窗体上绘制画面时，一般不会直接绘制窗体的 Canvas，这样可能会带来某些不必要的麻烦。C++ Builder 提供的 TPaintBox 组件是一个很好的作画对象，在窗体上作画一般通过

TPaintBox 组件得以实现。

TPaintBox 组件是 TCanvas 画布类的直接封装，但它确定了画布的位置和大小。TPaintBox 组件的另一个特点是：如果不在上面作画的话，它永远是透明的。

2. TTimer 组件

TTimer 组件在 C++ Builder 编程中被广泛使用，它封装了 Windows 的 WM_Timer 消息及其处理。TTimer 组件的地位非常重要，这个组件是一个定时器，它会在给定的一个间隔后发出一个 OnTimer 事件，这正好适合编制动画程序的需要。在程序中只需给定一个时间间隔，并在 OnTimer 事件的响应函数中画一帧画面，动画就执行了一步。

以下介绍 TTimer 组件的属性和事件。

• Enabled 属性。

声明：__property bool Enabled = {read=FEnabled, write=SetEnabled, default=1};

此属性用于打开和关闭计时器，关闭计时器后，程序不再收到时钟消息。

• Interval 属性。

声明：__property Cardinal Interval = {read=FInterval, write=SetInterval, default=1000};

决定 OnTimer 事件发生的间隔，单位是毫秒。如果此属性设为 0，则不发生 OnTimer 事件。

• OnTimer 事件。

声明：__property Classes::TNotifyEvent OnTimer = {read=FOnTimer, write=SetOnTimer};

当 Interval 属性指定的事件经过后，此事件触发。

OnTimer 事件将 WM_TIMER 事件发给一个窗口过程，但它有一个弊病——精确度不够高，在 Windows 98/95 操作系统中，WM_TIMER 事件发生的间隔最少也有 55ms，在 Windows NT 中也不能低于 10ms。

在动画程序中，为了加快显示速度，要求每帧之间有较小的时间间隔，TTimer 组件在较高要求下是不能满足的，这时可以考虑使用多线程或者多媒体定时器。有关这方面的内容将在本章的最后作以讨论。

6.1.3 生成高性能动画

在生成动画过程中，TCanvas 类的 CopyRect 方法是一个关键的方法。它用于内存位图和屏幕间频繁的数据交换，同时 TCanvas 类 Draw 和 StretchDraw 方法在动画编程中也是常用的。

我们先来看看精灵动画具体是如何实现的，为此编写一个 RocketAnim 的动画程序。虽然在程序中使用了一些特别的技术，如掩码位图（以下简称掩图），但其实现动画的基本原理遵循上述双缓冲实现动画的思想。在 RocketAnim 程序描述一个火箭在太空中飞行的场景，用户可以即时的调整火箭横向运动和纵向运动的速度，如图 6-1 所示。四个位图在动画处理时起了关键性的作用。



图 6-1 RocketAnim 演示程序

```
Graphics::TBitmap *Rocket; // 精灵位图
Graphics::TBitmap *Mask;   // 精灵掩图
Graphics::TBitmap *BG;     // 内存缓冲位图
Graphics::TBitmap *tmp;    // 填补位图
```

同时包含以下变量的声明：

```
int SpeedX,SpeedY;         // X 向速度和 Y 向速度
int XDir,YDir;             // 方向变量
TPoint Pos;                // 当前精灵位置
TRect mRect;               // 精灵位图矩形
```

动画生成需要依赖一个定时器组件，在本程序中的 TTimer 组件的 Interval 属性设为 50ms。

制作动画的初始化工作代码如下：

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Rocket=new Graphics::TBitmap();
    Mask=new Graphics::TBitmap();
    BG=new Graphics::TBitmap();
    tmp=new Graphics::TBitmap();
    Rocket->LoadFromFile("Rocket.bmp");
    Mask->LoadFromFile("Mask.bmp");
    BG->LoadFromFile("BG.bmp");
    tmp->Width=Rocket->Width;
```

```
        tmp->Height=Rocket->Height;
        SpeedX=TrackBar1->Position;
        SpeedY=TrackBar2->Position;
        XDir=1;YDir=1;
        Pos=Point(0,0);
        mRect=Rect(0,0,Rocket->Width,Rocket->Height);
        DrawObject();
        ShowObject();
}
```

生成动画的基本步骤是：填补上一次的作画区域——移动精灵——在内存中绘制精灵——显示一帧画面，这四步体现在了 OnTimer 事件的响应函数中。

Source 生成精灵动画相关代码

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
        PatchIt();          // 恢复作画区域
        MoveObject();   // 移动精灵
        DrawObject();    // 绘制精灵
        ShowObject();   // 显示画面
}
void __fastcall TForm1::PatchIt()
{
        TRect lRect=Rect(Pos.x,Pos.y,Pos.x+Rocket->Width,
                Pos.y+Rocket->Height);
        BG->Canvas->CopyMode = cmSrcCopy;
        BG->Canvas->CopyRect(lRect,tmp->Canvas,mRect);
}
void __fastcall TForm1::MoveObject()
{
        Pos.x+=SpeedX*XDir;
        Pos.y+=SpeedY*YDir;
        if (Pos.x<0||Pos.x>BG->Width-mRect.Right){
                XDir*=-1;
                if (Pos.x<0)
                        Pos.x*=-1;
                else
                        Pos.x=2*BG->Width-2*mRect.Right-Pos.x;
        }
        if (Pos.y<0||Pos.y>BG->Height-mRect.Bottom){
```

```
            YDir*=-1;
            if (Pos.y<0)
                Pos.y*=-1;
            else
                Pos.y=2*BG->Height-2*mRect.Bottom-Pos.y;
        }
}
void __fastcall TForm1::DrawObject()
{
        TRect lRect=Rect(Pos.x,Pos.y,Pos.x+Rocket->Width,
            Pos.y+Rocket->Height);
        tmp->Canvas->CopyRect(mRect,BG->Canvas,lRect);
        BG->Canvas->CopyMode=cmSrcAnd;
        BG->Canvas->CopyRect(lRect,Mask->Canvas,mRect);
        BG->Canvas->CopyMode=cmSrcPaint;
        BG->Canvas->CopyRect(lRect,Rocket->Canvas,mRect);
}
void __fastcall TForm1::ShowObject()
{
        PaintBox1->Canvas->Draw(0,0,BG); // 在 PaintBox 上显示。
}
```

这个程序执行速度是很快的，并且看不到任何闪烁。同时位图中火箭周围的黑色部分并没有被显示出来，而是透出了背景，实现这种效果是因为利用了“掩图”技术。

6.1.4 掩图技术

RocketAnim 程序中的火箭的边缘是不规则的。在 C++ Builder 中不提供多边形拷贝图像的函数，所以无法在图像上指定复制火箭的部分而去掉火箭周围不需要的部分。但利用掩图技术可以解决这个问题。

1. 拷贝模式

掩图技术不能再使用 CopyRect 方法的默认模式进行图像复制，设置 TCanvas 类对象的 CopyMode 属性可以改变 CopyRect 方法的拷贝模式，其取值如表 6-1 所列。

表 6-1 拷贝模式说明

取 值	运 算	含 义
cmSrcCopy	Source	默认的拷贝模式，拷贝结果是前景位图的颜色
cmBlackness	Black	用黑色填充拷贝区域
cmDstInvert	~Desk	反转背景位图颜色，忽略前景位图

（续）

取 值	运 算	含 义
cmMergeCopy	Desk&Source	使用按位与运算（And）组合前景位图和背景位图的颜色
cmMergePaint	Desk|~Source	使用按位或运算（Or）组合前景位图的反色和背景位图颜色
cmNotSrcCopy	~Source	复制前景位图的反像
cmNotSrcErase	~(Desk|Source)	使用按位或运算（Or）组合前景位图和背景位图的颜色，并反转结果
cmSrcAnd	Desk&Source	使用按位与运算（And）组合前景位图和背景位图的颜色
cmSrcErase	~Desk&Source	使用按位与运算（And）组合前景位图颜色和背景位图反色
cmSrcInvert	Desk^Source	使用按位异或运算（Xor）组合前景位图和背景位图的颜色
cmSrcPaint	Desk|Source	使用按位或运算（Or）组合前景位图和背景位图的颜色
cmWhiteness	White	用白色填充拷贝区域

设置了特殊的拷贝模式后，使用 CopyRect 方法的最终结果是前景位图和背景位图对应像素点进行某种模式运算后的结果。

2. 原图、掩图和背景

RocketAnim 程序不仅用到了一个火箭的图片，而且使用了一个与火箭有相同形状的掩图，RocketAnim 程序使用的位图如图 6-2 所示。



图 6-2 背景位图，火箭原图和火箭掩图

为了能够透出背景，需要对原图和掩图进行特殊处理：原图中火箭周围要屏蔽掉被设成黑色，而掩图中火箭的部分被设为黑色，而火箭周围的部分设为白色。黑色和白色是两种特殊的颜色，白色的颜色值是 0xFFFFFF，而黑色的颜色只是 0x000000，这样白色对某色进行与运算（And）、黑色对某色进行或运算（Or）的结果是原色，白色与某色进行或运算的结果是白色，黑色与某色进行与运算的结果是黑色。我们现在来回顾一下 RocketAnim 程序，在绘制火箭时，首先采用了与模式（cmSrcAnd）绘制掩图，而后使用或模式（cmSrcPaint）绘制原图。图 6-3 分析了绘制过程。

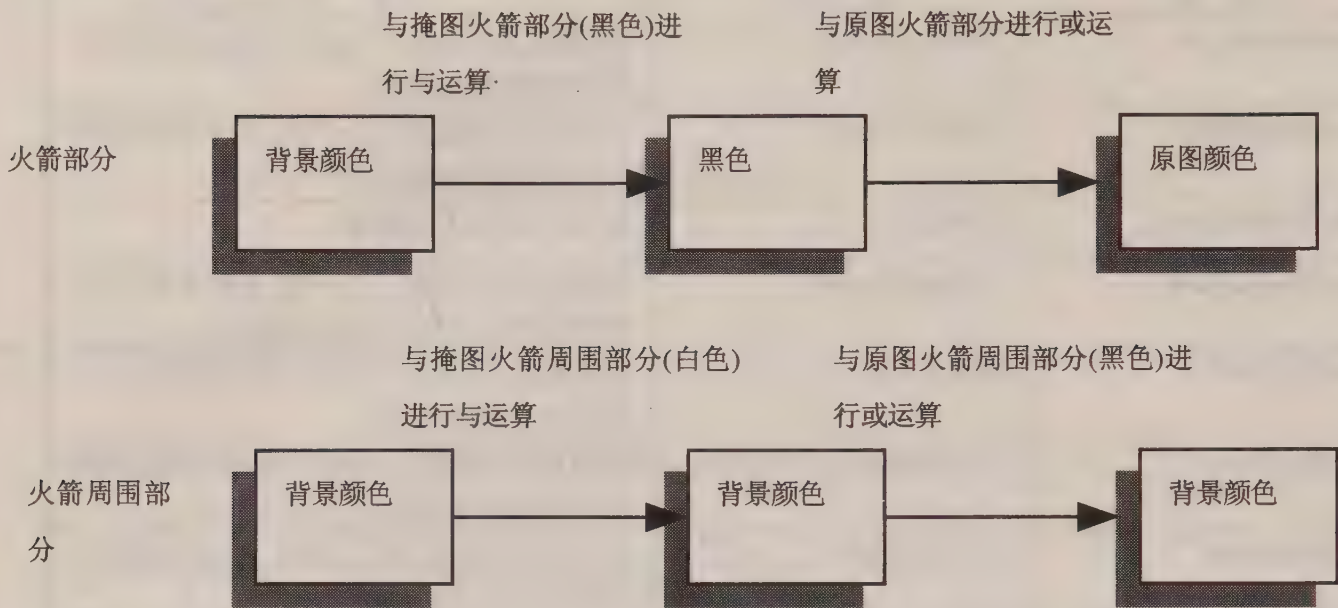


图 6-3　利用掩图技术绘制动画的工作原理

通过两次运算将火箭和背景都保留了下来。也可以使用另一种方案，即将原图火箭外面的部分设为白色，将掩图火箭部分设成白色，掩图火箭外面的部分设为黑色。绘制时第一次使用或模式复制掩图，而后使用与模式复制原图，同样能够达到相同效果。

6.2　实例创建分析

"BCB 屏幕保护程序"是在两种模式下工作的。一种是动画模式，它允许用户给定一张图片（如果用户给出多个图片，将随机抽出一张），并使这张图片在屏幕来回运动。另一种是变换模式，它将把用户指定的图片一张一张的显示出来，图片切换过程采用了多种不同的特效。

要实现一个完整的屏幕保护程序有很多工作要做，屏保真正运行起来的效果是一方面，同时屏保应该可以在预览模式下工作，可以设置对话框和用户密码。在"BCB 屏幕保护程序"中，忽略了预览支持和密码支持的部分，但它仍不失为一个能够正常工作的屏幕保护程序。

除了动画技术和特技显示的内容之外，作为一个屏幕保护程序，不可避免地涉及一些关键性的 C++ Builder 编程技术，包括程序参数的获取和相关处理、消息映射和处理 Windows 消息，在程序中操作注册表等等。鉴于这些技术的重要性，也将作为本章的一个重点讲述。

屏幕保护的设置界面如图 6-4 所示。

具体来说，实现"BCB 屏幕保护程序"要逐步完成以下工作：

- 获取并处理应用程序参数，并使屏保程序能够根据执行参数的不同启动不同的窗体。
- 实现屏幕保护程序框架，处理必要的 Windows 消息。
- 实现屏保程序在动画模式和变换模式下工作。
- 创建屏保设置窗体，实现设置屏保的相关代码。

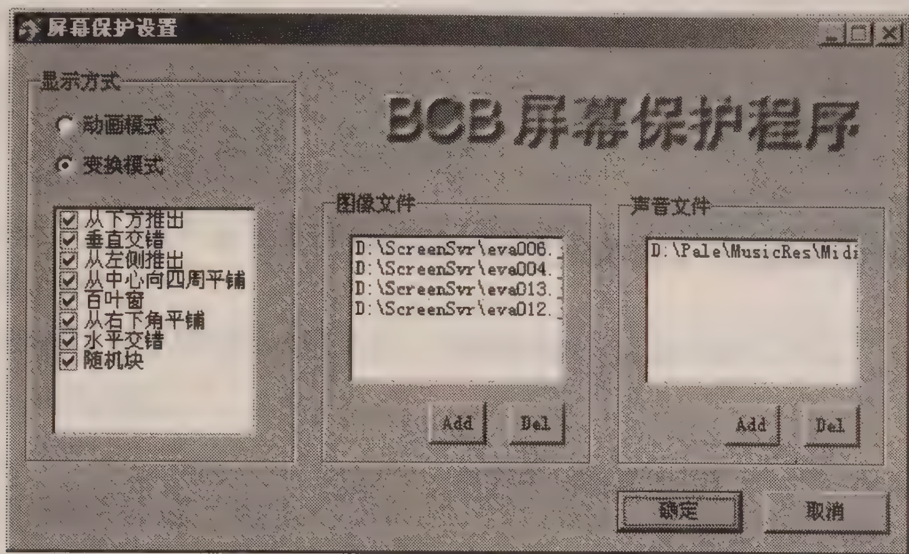


图 6-4　BCB 屏幕保护程序设置界面

- 播放音乐的辅助功能。

ScreenSvr 工程中主要包含下面的一些文件：

- 屏保窗体单元。TSvrForm，文件：Unit1.cpp，Unit1.h。
- 屏保设置窗体单元。类名：TCfgForm，文件：Unit1.cpp，Unit1.h。

另外，本范例中主程序文件 ScreenSvr.cpp 也需要进行处理。

6.3　实现屏幕保护程序框架

Windows 的屏幕保护程序以.scr 为扩展名，但本质上它是标准的 EXE 可执行文件。Windows 的屏幕保护程序有一些特性：首先，完整的屏幕保护程序并非是在一个模式下工作的，Windows 系统在调用屏保程序时会传递一些参数，不同的参数告诉屏保程序是执行全屏显示方式、预览方式还是屏保设置方式；其次，屏保程序必须处理一些特定的 Windows 消息。

6.3.1　获取并处理应用程序参数

在标准的 C 语言编程中， 通过 main 函数可以容易的接受用户传递的参数。例如下面的 C 的 main 函数的定义。

```
int main(int argc, char* argv[])
```

Windows 应用程序同样可以以命令行的方式执行，并传递参数。例如可以键入“NotePad C:\Demo.txt”运行记事本程序并用记事本打开 Demo.txt 文件。Windows 的屏幕保护程序也是带参数执行的，Windows 使用/s 参数全屏执行屏保程序，使用/c 参数进行屏保的设置，使用/a 参数预览屏保，使用/p 参数设置屏保密码。

C++ Builder 提供两个函数——ParamCount 和 ParamStr，用于获取并处理应用程序参数，通过 ParamCount 函数可以获得传递参数的具体数目，而通过 ParamStr 可以轻松的访问每一个参数。例如：ParamStr(1)可以从命令行获得第一个参数，如使用 ParamStr(0)则会返回程序本身的路径和文件名（例如，D:\Test\ScreenSvr.exe）。

“BCB 屏幕保护程序”省略了预览功能和密码部分，但是程序仍然需要在全屏显示和屏保设置部分两种不同工作方式下进行，这两种工作方式对应两个不同的窗体。我们希望根据接受参数的不同，执行相应的某一个窗体。这就必须手工修改主程序文件 ScreenSvr.cpp（假设已经完成了 TSvrForm 和 TCfgForm）。

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Title = "BCB 屏幕保护程序";
        if (ParamStr(1)=="/s"){
            Application->Initialize();
            Application->CreateForm(__classid(TSvrForm),&SvrForm);
            Application->Run();
        }
        else if (ParamStr(1)!="/a"&&ParamStr(1)!="/p"){
            Application->Initialize();
            Application->CreateForm(__classid(TCfgForm),&CfgForm);
            Application->Run();
        }
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
```

这里根据不同的传递参数创建不同的窗体。C++ Builder 的 TApplication 类的规则是第一个被建立的窗体是主窗体，所以不必担心菜单【Project/Options】选项中主窗体设置为 CfgForm 还是 SvrForm。

6.3.2 消息映射

当屏幕保护程序以非设置的方式执行时，有两个事件是必须处理的：其一，用户按下键盘或操作鼠标，这时执行退出屏保程序；其二，屏保运行时，有第二个屏保副本开始运行，这时需要避免屏保副本运行。

以上两种情况分别代表了 C++ Builder 中两种不同类型的 Windows 消息，一类是像鼠标消息、键盘消息等被直接作为事件封装在组件内，这类消息较为容易处理，只需要直接响应组件的响应事件即可。但另一类 Windows 消息并没有封装在组件内（其实也无法封装），此时必须采用一种特殊的机制——消息映射来决定应用程序如何响应特定的消息。

1. 拦截 Windows 消息

C++ Builder 允许程序员自己拦截并处理 Windows 消息，大致做法与 Visual C++或 Borladn C++编程一样。处理 Windows 消息需要以下几步：

(1) 建立消息映射表，把某条消息的处理权交给自定义的消息处理函数。

```
BEGIN_MESSAGE_MAP
      MESSAGE_HANDLER(Windows 消息名，TMessage，消息处理函数名)
      MESSAGE_HANDLER(...)
END_MESSAGE_MAP(TForm)
```

(2) 在程序中声明消息处理函数。

```
private:    // User declarations
void __fastcall 消息处理函数名(TMessage &Message);
```

(3) 在程序文件中编写消息处理函数，实现 Windows 消息的处理。例如：

```
void __fastcall MainForm:: 消息处理函数名(TMessage &Message)
{
      // TODO: Add your source code here
      MainForm->Dispatch(&Message);
}
```

2. 消息映射宏

定义消息映射表，采用了消息映射宏，这个宏是这样定义的：

```
#define BEGIN_MESSAGE_MAP virtual void __fastcall Dispatch(void *Message)     \
{     \
      switch  (((PMessage)Message)->Msg)                    \
      {
#define VCL_MESSAGE_HANDLER(msg,type,meth)                \
      case   msg:      \
      meth(*((type *)Message)); \
      break;
#define MESSAGE_HANDLER  VCL_MESSAGE_HANDLER
#define END_MESSAGE_MAP(base)  default:      \
      base::Dispatch(Message);            \
      break;       \
      }     \
}
```

从上述定义可以了解消息映射的实现原理。BEGIN_MESSAGE_MAP 宏是消息分派的入口。MESSAGE_HANDLER 宏定义了响应消息的函数句柄，参数 msg 是拦截的特定消息，type 参数存储消息内容的参数类型，可以是通用的 TMessage，也可以采用 C++ Builder 位每个特定消息定义的特定消息结构如 TWMEraseBkgnd。参数 meth 即消息的响应处理函数，END_MESSAGE_MAP 宏将其余 Windows 消息分发。

BEGIN_MESSAGE_MAP、MESSAGE_HANDLER、END_MESSAGE_MAP 三个宏一定一起使用，实际上，将这些宏连起来展开，即可得到如下结果：

```
virtual void __fastcall Dispatch(void *Message)
{
    switch (((PMessage)Message)->Msg)
    {
    case WM_NCHITTEST:
        OnNcHitTest(*((TMessage *)Message));
        break;
    default:
        TForm::Dispatch(Message);
        break;
    }
}
```

很明显，这是典型的消息循环书写方式。

3. TMessage 类型和 Dispatch 方法

消息处理函数的参数是一个 TMessage 类型，该类型是 VCL 的预定义结构。声明如下：

```
struct TMessage
{
    Cardinal Msg; //消息
    union
    {
        struct
        {
            Word WParamLo;     // 字参数低位
            Word WParamHi;     // 字参数高位
            Word LParamLo;     // 长字参数低位
            Word LParamHi;     // 长字参数高位
            Word ResultLo;     // 消息结果低位
            Word ResultHi;     // 消息结果高位
        };
        struct
        {
            long WParam;   // 字参数
            long LParam;   // 长字参数
            long Result;   // 消息结果
        };
    };
};
```

TMessage 类型的定义使我们不仅能够访问 Windows 消息的 WParam，LParam 和 Result，还可以进入其低位和高位。

Dispatch 方法作用是让 Windows 消息继续传递下去，但必要的时候可以不写该句，使消息被完全拦截。在本例中处理的几个消息就是这样的例子。

4. 实例中的消息处理

表 6-2 是在 BCB 屏幕保护程序中处理的消息及作用。

表 6-2　BCB 屏幕保护程序中的消息处理

Windows 消息	处理方式	作　用
WM_RBUTTONDOWN/ WM_LBUTTONDOWN	响应事件 OnMouseDown	判断鼠标动作，若有，退出屏保
WM_MOUSEMOVE	响应事件 OnMouseMove	判断鼠标动作，若有，退出屏保
WM_KEYDOWN	响应事件 OnKeyDown	判断键盘动作，若有，退出屏保
WM_TIMER	响应事件 OnTimer	绘制一帧画面
WM_CREATE	响应事件 OnCreate	进行屏保运行前的初始化工作
WM_ERASEBKGND	消息映射	禁止重画时擦除背景，以减少闪烁
WM_ACTIVATE	消息映射	使屏保窗口在失去激活时退出
WM_SYSCOMMAND	消息映射	使屏幕保护程序的第二个副本不能运行

对于后三个不能直接处理的消息，在头文件 Unit1.h 中建立消息映射表。

```
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_ERASEBKGND,TWMEraseBkgnd,WMEraseBkgnd)
    MESSAGE_HANDLER(WM_ACTIVATE,TWMActivate,WMActivate)
    MESSAGE_HANDLER(WM_SYSCOMMAND,TWMSysCommand,WMSysCommand)
END_MESSAGE_MAP(TForm)
```

消息响应函数部分的代码如下：

```
void __fastcall TSvrForm::WMEraseBkgnd(TWMEraseBkgnd & Msg)
{    //禁止重画时擦除背景
    Msg.Result = false;
}
void __fastcall TSvrForm::WMActivate(TWMActivate & Msg)
{    //失去激活时退出
    if (Msg.Active==false) Close();
}
void __fastcall TSvrForm::WMSysCommand(TWMSysCommand & Msg)
{    //如果已有屏保程序运行，则不再运行
    if(Msg.CmdType==SC_SCREENSAVE)
        Msg.Result=true;
    else
        SvrForm->Dispatch(&Msg);
}
```

当用户键盘或鼠标有任何动作时，退出屏保程序。这可通过响应 FormSvr 窗体的事件完成。

```
void __fastcall TSvrForm::FormKeyDown(TObject *Sender, WORD &Key,TShiftState Shift)
{
        Close();
}
void __fastcall TSvrForm::FormMouseDown(TObject *Sender, TMouseButton Button,
        TShiftState Shift, int X, int Y)
{
        Close();
}
void __fastcall TSvrForm::FormMouseMove(TObject *Sender, TShiftState Shift,int X, int Y)
{
        int xDistance=X-aPoint.x;
        int yDistance=Y-aPoint.y;
        // 用户移动鼠标超过某个定值后退出
        if ((xDistance<-MouseMoveDistance)||(xDistance>MouseMoveDistance))
                Close();
        else if ((yDistance<-MouseMoveDistance)||(yDistance>MouseMoveDistance))
                Close();
}
```

6.4 动画和特技显示

6.4.1 屏保的动画部分

"BCB 屏幕保护程序"允许用户提供一组用于屏保显示的图片，这组图片存放在 PicList 文件中。在屏保的动画模式下，从这组图片中随机抽出一张，作为动画精灵。

这里的动画基本与 RocketAnim 程序中的动画一致，所不同的是将火箭换成了图片，同时也不需要使用掩图。

首先，在 OnCreate 函数中加入与动画模式相关的初始化代码。

```
Bit1=new Graphics::TBitmap;     // 内存缓冲位图
Bit2=new Graphics::TBitmap;     // 存放精灵图片
tmp=new Graphics::TBitmap;      // 临时位图
PicList=new TStringList();
AnsiString Path=ExtractFilePath(Application->ExeName);
PicList->LoadFromFile(Path+"PicList");        // 读入图片组信息
ReadReg();      // 对注册表信息函数，获得屏保模式以及其他信息
```

```
randomize();     //初始化随机数
//设置窗体
Width=Screen->Width;
Height=Screen->Height;
Cursor=crNone;
//获取并纪录鼠标位置
POINT  WinPoint;
GetCursorPos(&WinPoint);
aPoint.x=WinPoint.x;
aPoint.y=WinPoint.y;
if (Mode==0){
    //动画模式的初始化工作
    Bit1->Width=Width;
    Bit1->Height=Height;
    Bit1->Canvas->Brush->Color=clBlack;
    Bit1->Canvas->FillRect(Rect(0,0,Width,Height));
    //随机读入一张图片作为精灵
    Pic=floor(random(PicList->Count));
    LoadPic(PicList->Strings[Pic]);
    oRect=Rect(0,0,Bit2->Width,Bit2->Height);
    //设置精灵的速度、方向和初始位置
    xSpeed=ySpeed=2;
    xDir=yDir=1;
    cLocation.x=cLocation.y=0;
    oLocation=cLocation;
}
else {
    ......
```

这段代码中，ReadReg 函数将在下一节讲到注册表时建立，GetCursorPos 是一个 API 函数，它通过参数返回鼠标的当前坐标（以屏幕位坐标系），LoadPic 函数用于将指定图片读入对象位图 Bit2，这个函数建立如下：

```
void __fastcall TSvrForm::LoadPic(AnsiString FName)
{
    float Ratio,W,H; // 长宽和长宽比例
    TImage *tmpImage = new TImage(SvrForm); // 临时图像组件
    tmpImage->AutoSize=true;
    if (FileExists(FName)){
        tmpImage->Picture->LoadFromFile(FName);
        Ratio = float(tmpImage->Width)/float(tmpImage->Height);
        W=tmpImage->Width;H=tmpImage->Height;
```

```
            switch (Mode){
            case 0:     // 动画模式
            if (W <=600&&H<= 450)
            Bit2->Assign(tmpImage->Picture);
            else if (Ratio>4/3){
            Bit2->Width=600;
            Bit2->Height=600/Ratio;
            Bit2->Canvas->StretchDraw(Rect(0,0,Bit2->Width,Bit2->Height),
                        tmpImage->Picture->Graphic);
                }
            else {
            Bit2->Height=450;
            Bit2->Width=450*Ratio;
            Bit2->Canvas->StretchDraw(Rect(0,0,Bit2->Width,Bit2->Height),
                        tmpImage->Picture->Graphic);
            }
            break;
            case 1:     // 变换模式
            Bit2->Canvas->Brush->Color=clBlack;
            Bit2->Canvas->FillRect(Rect(0,0,600,450));
            if (W <=600&&H<= 450)
            Bit2->Canvas->Draw((600-W)/2,450-H)/2,
                        tmpImage->Picture->Graphic);
            else{
            if (W>600){
                        W=600;    H=600 / Ratio;
            }
            if (H >450){
                        H = 450;    W = H * Ratio;
            }
            Bit2->Canvas->StretchDraw(Rect((600-floor(W))/2,(450-floor(H))/2,
                        (600-floor(W))/2+floor(W),(450-floor(H))/2+floor(H)),
                  tmpImage->Picture->Graphic);
            }
        }
        }
        delete tmpImage;
    }
```

BmpW 和 BmpH 是程序中定义的最大宽度和最大高度的常量。LoadPic 函数采取按比例放缩的方法处理尺寸过大的图像。对于动画模式和变换模式有所不同，动画模式中，对象位图

的宽和高永远与图片（或按比例放缩的图片）的宽和高一致；而变换模式中，对象位图的宽和高永远和最大宽度和最大高度一致，没有图片的地方用黑色填充。

动画实现仍然遵循“填补—移动—绘制—显示”四步，分别在以下四个函数内实现。

Source 实例程序中绘制动画的相关代码

```
void TSvrForm::PatchBkgnd(void)
{
        // 恢复作画区域
        Bit1->Canvas->Brush->Color=clBlack;
        Bit1->Canvas->FillRect(lRect);
}
void TSvrForm::DrawObject(void)
{
        // 绘制精灵图片
        Bit1->Canvas->CopyRect(lRect,Bit2->Canvas,oRect);
}
void TSvrForm::MoveObject(void)
{
        // 移动精灵
        oLocation=cLocation;
        cLocation.x+=xSpeed*xDir;
        cLocation.y+=ySpeed*yDir;
        if (cLocation.x<0||cLocation.x>Bit1->Width-oRect.Right)
        {
                xDir*=-1;
                if(cLocation.x<0)
                        cLocation.x*=-1;
                else
                        cLocation.x=2*Bit1->Width-2*oRect.Right-cLocation.x;
        }
        if (cLocation.y<0||cLocation.y>Bit1->Height-oRect.Bottom)
        {
                yDir*=-1;
                if(cLocation.y<0)
                        cLocation.y*=-1;
                else
                        cLocation.y=2*Bit1->Height-2*oRect.Bottom-cLocation.y;
        }
        lRect=Rect( cLocation.x,cLocation.y,
```

```
            cLocation.x+Bit2->Width,
            cLocation.y+Bit2->Height);
}
void TSvrForm::ShowObject(void)
{
    // 显示画面
    TRect oRect=Rect(oLocation.x,oLocation.y,
                oLocation.x+Bit2->Width,
                oLocation.y+Bit2->Height);
    Canvas->CopyRect(oRect,Bit1->Canvas,oRect);
    Canvas->CopyRect(lRect,Bit1->Canvas,lRect);
}
```

最后在 OnTimer 事件的响应函数中依次执行四个函数，就得到图片动画效果。

6.4.2 技巧显示

图形的技巧显示可以经常看到，Mircosoft PowerPoint 软件中提供的幻灯片切换效果就是一例。图形的技巧显示的实质是：按照某种规则，将图像的不同部分以某种次序一步步显示出来，直至图像完全显示。各种技巧显示的原理并不复杂。

技巧显示中常用的函数是 TCanvas 类的 CopyRect 方法，其作用是将对象图片的某一部分绘制到屏幕的指定位置。

进行技巧显示的前期工作是在 OnCreate 函数中加入初始化代码，这里关键的变量有三个：ChangePic——布尔型，决定是否更换图片；Pic——目前显示的图形序号；flag——目前显示效果的序号。

在 OnTimer 的响应函数中加入代码，执行具体的特技显示。

```
void __fastcall TSvrForm::Timer1Timer(TObject *Sender)
{
    switch (Mode){
    case 0:
        ......
    case 1:
        if (ChangePic){
            // 更换显示方式。
            do {
                flag=(flag+1)%8;
            }while (!Effect[flag]);
            // 更换图片。
            Pic=(Pic+1)%PicList->Count;
            LoadPic(PicList->Strings[Pic]);
```

```
                ChangePic=false;
                x=0;
                if (flag==7)
                    ZeroMemory(b,sizeof(b));
            }
            PlayPic();
        }
    }
```

当一张图片显示完成后，ChangePic 变量被赋值 true，在 OnTimer 事件中进行更画图片并更换显示方式的工作。本程序允许用户选择使用哪些显示方式。从注册表中读入用户所选显示方式，并存入 Effect 数组中。

PlayPic 函数完成具体的特技显示。本程序目前共支持 8 种特效，包括从下方推出、垂直交错、从左侧推出、从中心向四周平铺、百叶窗、从右下角平铺、水平交错、随机块。现简单讲解其中几种的实现原理。

1. 从下方推出

将对象位图的第一条水平线显示在背景位图的最下方，后依次将前两条、前三条显示在背景位图的下方，直到对象位图完全显示为止。其效果是显示的位图由下而上浮起。同理可以实现上下左右四个方向的推拉。

程序算法：

```
switch (flag){
case 0: // 从下方推出
    if (x<BmpH){
        Bit1->Canvas->CopyRect(Rect(0,BmpH-x,BmpW,BmpH),
            Bit2->Canvas,Rect(0,0,BmpW,x));
        Canvas->Draw(Lft,Tp,Bit1);
        x+=2;
    }
    else ChangePic=true;
    break;
```

2. 垂直交错

将对象位图分为两部分显示，奇数条扫描线由上往下搬移，偶数条扫描线由下往上搬移，两者同时进行。其效果是背景位图上下两端出现较淡图形向屏幕中央移动，直到完全清楚为止。类似可以实现水平交错。

程序算法：

```
switch (flag){
......
case 1: // 垂直交错
    if (x<=BmpH){
```

```
        y=x;
        while (y >0){
              Bit1->Canvas->CopyRect(Rect(0,y-1,BmpW,y),
                    Bit2->Canvas,
                    Rect(0,BmpH-x+y-1,BmpW,BmpH-x+y));
              Bit1->Canvas->CopyRect(Rect(0,BmpH-y,BmpW,BmpH-y+1),
                    Bit2->Canvas,Rect(0,x-y,BmpW,x-y+1));
              y-=2;
        }
        Canvas->Draw(Lft,Tp,Bit1);
        x+=2;
  }
  else ChangePic=true;
  break;
```

3. 纵向百叶窗

将对象位图纵向分为 N 组显示，依次将每组的第一列、第二列、第三列像素显示在背景位图的对应位置，直到全部完成为止，如图 6-5 所示。同理可以实现横向百叶窗。

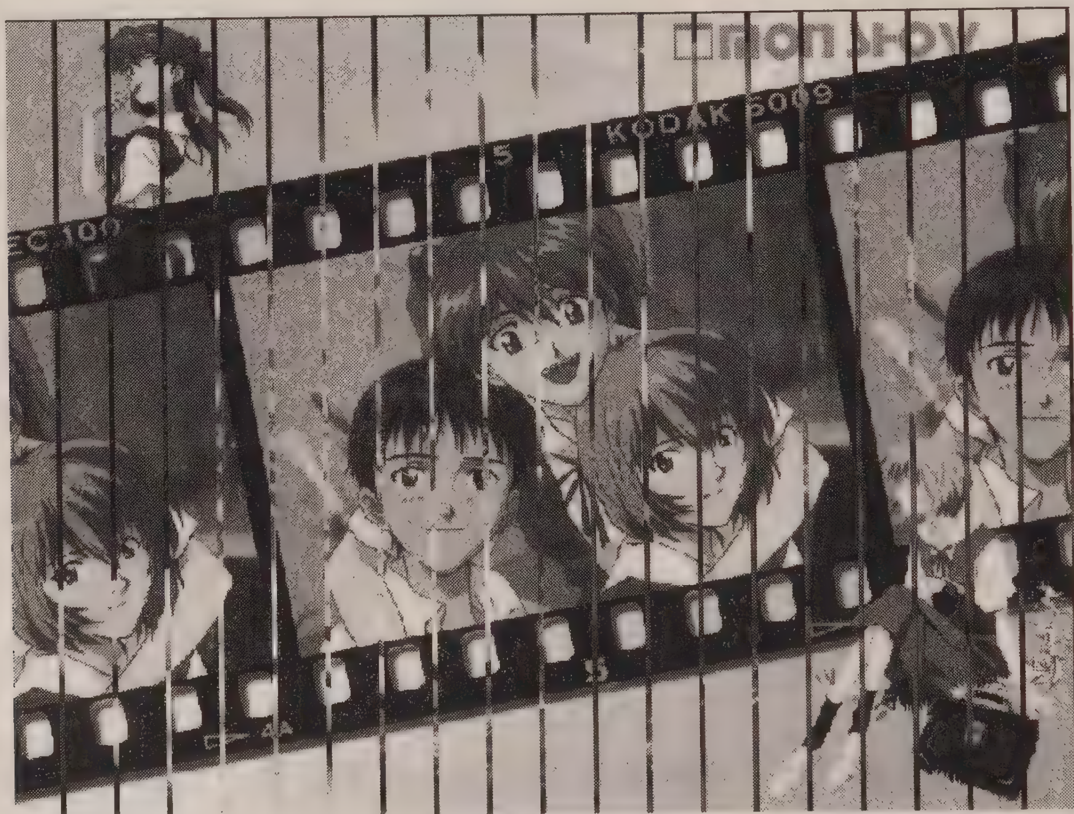


图 6-5　纵向百叶窗显示

程序算法：

```
switch (flag){
......
case 4:  // 百叶窗
      int w=int((BmpW+19)/20);  // 每叶的宽度
```

```
    if (x<=w){
        for (y=0;y<20;y++){ // 分为 20 叶
            Bit1->Canvas->CopyRect(Rect(y*w+x-1,0,y*w+x,BmpH),
                    Bit2->Canvas,Rect(y*w+x-1,0,y*w+x,BmpH));
        }
        Canvas->Draw(Lft,Tp,Bit1);
        x++;
    }
    else  ChangePic=true;
    break;
```

4. 随机块

将对象位图分成多个方形区域，随机将每一个方形区域显示在背景位图的对应位置，直到图像完全显示出，如图 6-6 所示。同理可以实现横向随机条显示，纵向随机条显示。



图 6-6　随机块特技显示

程序算法：

```
switch (flag){
……
case 7: // 随机块
    if (x<300){
        int i,j;
        while (b[i=random(15)][j=random(20)]);
        Bit1->Canvas->CopyRect(Rect(30*j,30*i,30*j+30,30*i+30),
            Bit2->Canvas,Rect(30*j,30*i,30*j+30,30*i+30));
```

```
            Canvas->Draw(Lft,Tp,Bit1);
            b[i][j]=true; // 纪录某一块是否已经显示的数组
            x++;
        }
        else ChangePic=true;
        break;
```

其他未列出技巧显示请参看源程序。

6.4.3 音乐播放功能

很多屏幕保护程序在显示画面的同时播放背景音乐，在 BCB 屏保程序中也将实现这个功能。读者可以看到，借助 C++ Builder 提供的 TmediaPlayer 组件实现这一点有多么容易。

关于 TMediaPlayer 以及 C++ Builder 多媒体技术的详细内容，将在本书第 7 章详细介绍，这里将仅实现简单的音乐播放功能。

程序支持用户指定一组音乐循环播放，音乐文件信息包含在 SndList 文件中。

首先将组件选项板中 System 页的 TMediaPlayer 组件放到屏保窗体上，设置其 Visible 属性为 false。

在 OnCreate 函数中加入初始化代码。

```
SndList=new TStringList();
AnsiString Path=ExtractFilePath(Application->ExeName);
SndList->LoadFromFile(Path+"SndList");
Snd=0;
if (SndList->Count>0){  //打开媒体播放机，播放第一首乐曲
    MediaPlayer1->FileName=SndList->Strings[Snd];
    MediaPlayer1->Open();
    MediaPlayer1->Play();
}
```

响应 TMediaPlayer 组件的 OnNotify 事件，在一首乐曲结束后播放下一首。

```
void __fastcall TSvrForm::MediaPlayer1Notify(TObject *Sender)
{
    if (MediaPlayer1->NotifyValue==nvSuccessful){
        Snd=(Snd+1)%SndList->Count; // 循环播放
        MediaPlayer1->FileName=SndList->Strings[Snd];
        MediaPlayer1->Open();
        MediaPlayer1->Play();
    }
}
```

这就是实现背景音乐循环播放的全部工作。

6.5 屏保设置部分的实现

一个完整的屏幕保护程序应该为用户提供设置功能。Windows 95/98 中选择屏保时的“设置”选项启动一个屏幕保护程序的设置模式。

在范例程序“BCB 屏幕保护程序”中，提供给用户设置以下选项：

- 显示图像文件。
- 播放背景音乐文件。
- 屏保工作模式，包括选择动画模式或是变换模式以及采用哪些特技显示。

屏保设置窗体如图 6-7 所示，它的主要部分包括两个 RadioButton、两个 ListBox 列表框和一个 CheckListBox 检查列表框。

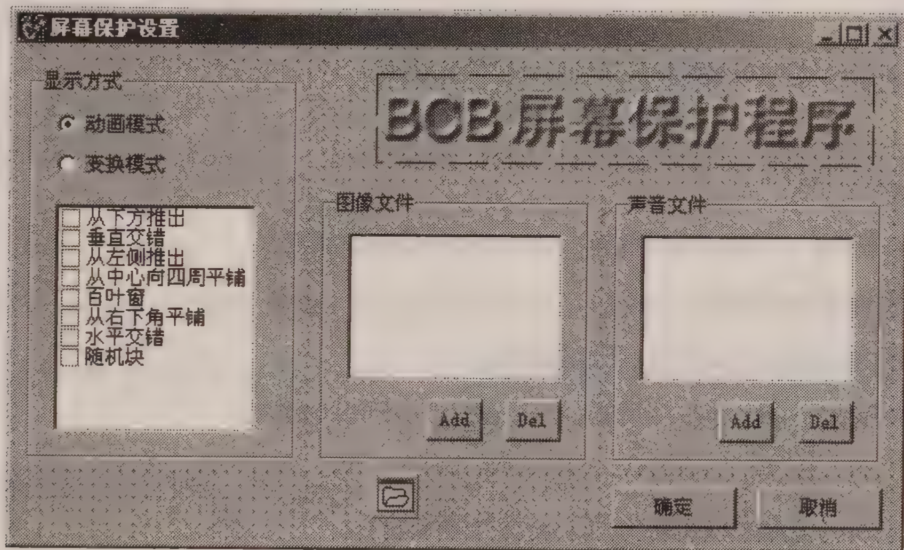


图 6-7 设计“BCB 屏幕保护程序”设置界面

TCheckListBox 组件

TCheckListBox 检查列表框组件类似于列表框组件，但它的每个项目前存在一个检查方框，方框中打勾表示项目被选中。TCheckListBox 继承于 TCustomListBox 类。

- AllowGrayed 属性。

声明：__property bool AllowGrayed = {read=FAllowGrayed, write=FAllowGrayed, default=0};

决定检查方框可否有“灰色”状态。如果此值为真，每个项目存在选中、未选中、灰色三种状态。

- Checked 属性。

声明：__property bool Checked[int Index] = {read=GetChecked, write=SetChecked};

描述检查列表框中每个项目的选择状态。选中时为真值，未选中、灰色时为假值。

- State 属性。

声明：__property Stdctrls::TCheckBoxState State[int Index] = {read=GetState, write=SetState};

描述检查列表框中每个项目的选择状态。有三种可能取值：cbUnchecked, cbChecked, cbGrayed。

6.5.1　存取文件列表

图像文件和背景音乐文件通过两个列表框选择。通过 TListBox 的 LoadFromFile 方法和 SaveToFile 方法，实现图像文件列表和背景音乐文件列表的存取是很方便的。

在 TCfgForm 的 OnCreate 函数中读入上次的设定

```
try{
    AnsiString Path=ExtractFilePath(Application->ExeName);
    ListBox1->Items->LoadFromFile(Path+"PicList");
    ListBox2->Items->LoadFromFile(Path+"SndList");
}
```

实现增加文件到列表框。

```
void __fastcall TCfgForm::Button3Click(TObject *Sender)
{
    // 增加图像文件
    OpenDialog1->Filter = GraphicFilter(__classid(TGraphic));
    if (OpenDialog1->Execute())
        if (FileExists(OpenDialog1->FileName))
            ListBox1->Items->Add(OpenDialog1->FileName);
}
void __fastcall TCfgForm::Button6Click(TObject *Sender)
{
    // 增加背景音乐文件
    OpenDialog1->Filter =
        "All Music Files(*.mdi;*.wav)|*.mid;*.wav|MIDI Files(*.mdi)|*.mid|Wave Files(*.wav)|*.wav";
    if (OpenDialog1->Execute())
        if (FileExists(OpenDialog1->FileName))
            ListBox2->Items->Add(OpenDialog1->FileName);
}
```

当用户确定设置时，将两个列表框中的文件保存。在 Button1Click 响应函数中加入：

```
AnsiString Path=ExtractFilePath(Application->ExeName);
ListBox1->Items->SaveToFile(Path+"PicList");
ListBox2->Items->SaveToFile(Path+"SndList");
```

6.5.2　使用注册表

设置屏保工作模式时，我们演示了不同的存储模式——按照 Win32 应用程序的习惯，利用 Windows 注册表来存取信息。

1. Win32 注册表

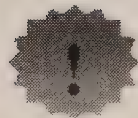
注册表是一个数据库，它存储着有关系统硬件配置、Windows 和 Windows 应用程序的信息。注册表以层次的格式进行组织，其结构十分复杂，含有多层主键、子键和值。但我们可以简单的将注册表理解为树性结构。

Windows 注册表包含六个根键，四个针对计算机信息的根键如下：

HKEY_CLASSES_ROOT	关于 Windows 用户界面、OLE、快捷方式的信息。
HKEY_LOCAL_MACHINE	计算机设备信息。包括安装的硬件和硬件驱动程序的设置。
HKEY_CURRENT_CONFIG	当前硬件信息。
HKEY_DYN_DATA	由即插即用例程使用的动态状态信息。

两个针对用户的根键如下：

HKEY_CURRENT_USER	当前计算机注册用户信息。
HKEY_USERS	用户信息。包括应用程序设置和桌面设置信息。



可以使用 Windows 95/98 中的 RegEdit 或 Windows NT 中的 RegEdit32 程序来浏览或编辑注册表的值。修改注册表必须谨慎小心，因为错误的修改很可能会导致系统崩溃。

2. TRegistry 类

Windows 有一系列 API 函数用于同注册表进行交互。C++ Builder 的 TRegistry 类封装了这些函数，这使得操作注册表变得非常容易。

- CurrentKey 属性。

声明：__property HKEY CurrentKey = {read=FCurrentKey, nodefault};

返回当前打开的主键。

- CurrentPath 属性。

声明：__property AnsiString CurrentPath = {read=FCurrentPath};

返回当前打开的主键的路径。

- RootKey 属性。

声明：__property HKEY RootKey = {read=FRootKey, write=SetRootKey, nodefault};

设置或读取当前操作的根键。

- CreateKey 方法。

声明：bool __fastcall CreateKey(const System::AnsiString Key);

新建一个主键。为当前主键的一个子键。

- DeleteKey 方法。

声明：bool __fastcall DeleteKey(const System::AnsiString Key);

删除当前主键下的指定子主键。

- KeyExists 方法。

声明：bool __fastcall KeyExists(const System::AnsiString Key);

确定指定名称的主键是否存在。

- OpenKey 方法。

声明：bool __fastcall OpenKey(const System::AnsiString Key, bool CanCreate);

打开指定名称的操作主键。如果 CanCreate 参数为真，当指定主键不存在时，则新建此主键。

- ValueExists 方法。

声明：bool __fastcall ValueExists(const System::AnsiString Name);

确定当前主键下指定名称的值是否存在。

- WriteInteger 方法。

声明：void __fastcall WriteInteger(const System::AnsiString Name, int Value);

在当前主键下写入指定名称的整型值 。类似的方法有：WriteBinaryData、WriteBool、WriteCurrency、WriteDate、WriteDateTime、WriteExpandString、WriteFloat、WriteString、WriteTime。

- ReadInteger 方法。

声明：int __fastcall ReadInteger(const System::AnsiString Name);

读取当前主键下指定名称的整型值。类似的方法有：ReadBinaryData、ReadBool、ReadCurrency、ReadDate、ReadDateTime、ReadFloat、ReadString、ReadTime。

使用 TRegistry 类时应首先创建一个 TRegistry 类实例，然后使用 OpenKey 方法打开一个主键。再使用相应的方法读取或写入不同类型的值。

3. 使用注册表保存设置信息

一般来说，应用程序的信息写在 HKEY_CURRENT_USER\Software\部分中。“BCB 屏幕保护程序”在这个主键下建立 PaleSoft\BCBScreenSaver 主键用于存取屏保设置信息。图 6-8 显示了注册表中加入了“BCB 屏幕保护程序”的设置信息。

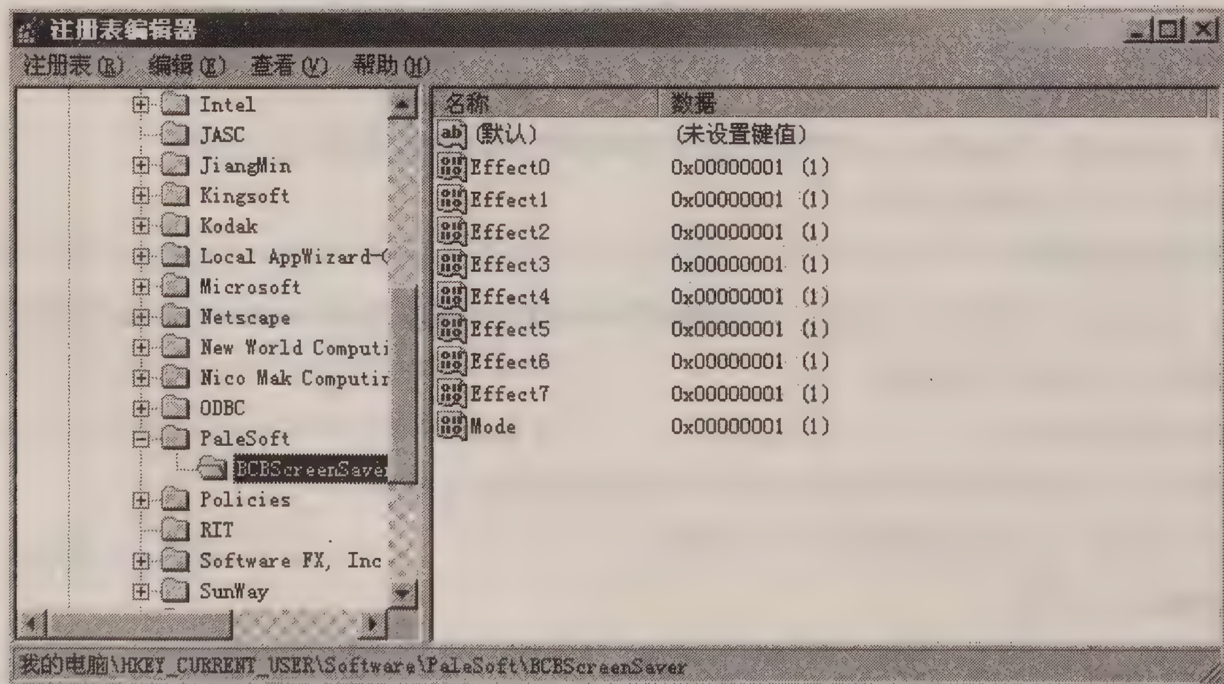


图 6-8 注册表加入了“BCB 屏幕保护程序”的设置信息

以下是 TCfgForm 中关于注册表操作的代码。

FormCreate 函数中：

```
Reg= new TRegistry();
Reg->RootKey=HKEY_CURRENT_USER;  // 选择操作根键
// 打开主键，如主键不存在，则新建此主键
Reg->OpenKey("\\Software\\PaleSoft\\BCBScreenSaver\\",true);
if (!Reg->ValueExists("Mode")){   // 判断主键下值是否存在，如不存在，
      Reg->WriteInteger("Mode",0);      // 则新建这组值。
      RadioButton1->Checked=true;
      for (int i=0;i<8;i++)
            Reg->WriteBool("Effect"+IntToStr(i),true);
}
else{
      // 从注册表读取值，并反映在设置窗体上。
      if(Reg->ReadInteger("Mode")==0)
            RadioButton1->Checked=true;
      else RadioButton2->Checked=true;
      for (int i=0;i<8;i++)
            CheckListBox1->Checked[i]=Reg->ReadBool("Effect"+IntToStr(i));
}
```

确定键响应函数中：

```
if (RadioButton1->Checked)
      Reg->WriteInteger("Mode",0);
else Reg->WriteInteger("Mode",1);
for (int i=0;i<8;i++)
      Reg->WriteBool("Effect"+IntToStr(i),
CheckListBox1->Checked[i]);
```

另一方面，在屏保窗体文件中从注册表读入这些信息，建立 ReadReg 函数。

```
void TSvrForm::ReadReg()
{
      Reg=new TRegistry();
      Reg->RootKey=HKEY_CURRENT_USER;
      Reg->OpenKey("\\Software\\PaleSoft\\BCBScreenSaver\\",true);
      if (!Reg->ValueExists("Mode")){
            Reg->WriteInteger("Mode",0);
            Mode=0;
            for (int i=0;i<8;i++)
                  Reg->WriteBool("Effect"+IntToStr(i),true);
      }
      else {
```

```
            Mode=Reg->ReadInteger("Mode");
            for (int i=0;i<8;i++)
                Effect[i]=Reg->ReadBool("Effect"+IntToStr(i));
        }
    }
```

6.6　动画技术的其他话题

可以确信，动画编程将在长时间内作为计算机编程的一个主题。前文中通过范例程序“BCB 屏幕保护程序”讲述了精灵动画技术和技巧显示，但动画编程的技术远远不止这些。本节将从实用角度出发，对其他动画应用和关于 Windows 动画的高级话题加以讨论。

6.6.1　桌面精灵动画

经常看到一些好玩的 Windows 桌面游戏，那些游戏中动画被画在了窗口外面，甚至遍布整个屏幕。在桌面上干净利落的绘制动画，这是怎么做到的？

实现桌面精灵动画有两个关键问题：

- 如何在窗口之外绘图。
- 如何保证其他窗体或桌面显示不受影响。

第一个问题容易实现，首先定义一个画布类实例：

```
TCanvas *Desk;
Desk =new TCanvas();
```

API 函数 GetDC 用于获得设备描述表句柄。声明如下：

```
HDC GetDC( HWND hWnd );
```

如果参数 hWnd 为 0，则可获得屏幕的设备描述表句柄。将此句柄赋给 Desk 的 Handle 属性后，就可以通过 Desk 来操作屏幕画布，从而也就可以在屏幕任意部分作画了。

```
Desk->Handle=GetDC(0);
```

回答第二问题并不太简单。首先，绘制精灵动画的四个步骤仍是必须的，但对于桌面精灵动画来说，仅仅这样做仍有可能会造成屏幕的混乱。精灵在走某一步时，记录下了填补区域的内容，但此时这块背景上的程序进行新内容绘制的操作或是被最小化了。但是下一步程序仍然会把不正确的内容填补在作过的区域，这就造成了难看的花屏。

解决这个问题可以使用 API 函数 RedrawWindow 告诉相应的窗口或桌面重画。RedrawWindow 用到 Windows 区域（HGN）技术的相关内容，具体来说，我们首先应确定需要重画的区域，而后通知该区域涉及的窗口进行该区域的重画操作。

以下是一个有趣的桌面精灵动画范例。该程序可以选择 15 种不同的角色，并且可以控制角色在桌面上自由移动。其中某个角色的位图如图 6-9 所示。

本程序演示另一种简便透出背景的技术。利用了 TBitmap 类的 TransparentColor 和 Transparent 两个属性。即，可以在绘制位图指定一种透明色，绘制时凡是遇到颜色为透明色的点则不进行绘制操作。前提是必须确定所画精灵周围区域是某种固定颜色，而精灵区域完全

没有这种颜色，这时再将 TransparentColor 设为这种颜色，并使用背景位图的 Draw 方法（注意不能再使用 CopyRect 方法，否则透明色设定无效）将对象为图画上去，就实现了与掩图技术一样的效果。

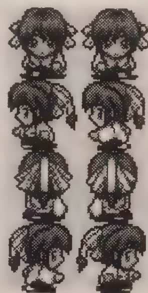


图 6-9　四个方向的精灵位图

以下是定时器组件的响应函数，包含实现桌面精灵动画的关键代码。

Source 实现桌面精灵动画的关键代码

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
      char TopTitle[20];
      HWND TopWindow;
      Pic=Frms[FrameOne*4+Dir];
      FrameOne=(FrameOne+1)%2;
      TopWindow=GetForegroundWindow();
      GetWindowText(TopWindow,TopTitle, sizeof(TopTitle));
      Desk->CopyRect(Rect(Lft,Tp,Lft+Pic->Width,Tp+Pic->Height),
            Recover->Canvas,Rect(0,0,Pic->Width,Pic->Height));
      HRGN oldRgn=CreateRectRgn(Lft,Tp,Lft+Pic->Width,Tp+Pic->Height);
      switch(Dir){
      case 2: Tp=(Tp-2)%Screen->Height;
            break;
      case 0: Tp=(Tp+2)%Screen->Height;
            break;
      case 1: Lft=(Lft-2)%Screen->Width;
            break;
      case 3: Lft=(Lft+2)%Screen->Width;
      }
      HRGN newRgn=CreateRectRgn(Lft,Tp,Lft+Pic->Width,Tp+Pic->Height);
      HRGN tmpRgn=CreateRectRgn(0,0,0,0);
      CombineRgn(tmpRgn,oldRgn,newRgn,RGN_DIFF);
      RedrawWindow(0,NULL,tmpRgn,RDW_ERASE|RDW_INVALIDATE|RDW_ALLCHILDREN);
      lRect=Rect(Lft,Tp,Lft+Pic->Width,Tp+Pic->Height);
```

```
        mRect=Rect(0,0,Pic->Width,Pic->Height);
        Recover->Canvas->CopyRect(mRect,Desk,lRect);
        Desk->Draw(Lft,Tp,Pic);
        DeleteObject(tmpRgn);
        DeleteObject(newRgn);
        DeleteObject(oldRgn);
}
```

这段代码里首先获得了当前应该显示的图像。而后计算了精灵新的位置，并且获得上一次和当前精灵所占的区域（RGN），并通过 CombineRgn 函数获得需要通知重画的区域。CombineRgn 函数如下声明：

```
int CombineRgn(
  HRGN hrgnDest,          // 目标区域句柄
  HRGN hrgnSrc1,          // 源区域 1 句柄
  HRGN hrgnSrc2,          //源区域 2 句柄
  int fnCombineMode       // 区域合并模式
 );
```

其中，fnCombineMode 可以取 RGN_AND、RGN_COPY、RGN_DIFF、RGN_OR、RGN_XOR 等值，分别表示两区域取交、拷贝源区域 1 到目标区域、属于区域 1 而不属于区域 2、两区域取和、两区域取和但不包括两区域的相交部分。特别需要说明的是，在执行 CombineRgn 函数时，hrgnDesk 必须已经存在，所以程序中“HRGN tmpRgn = CreateRectRgn (0,0,0,0);”是必须的。

之后调用 RedrawWindow 函数通知窗口刷新。RedrawWinow 函数声明如下：

```
BOOL RedrawWindow(
  HWND hWnd,                  // 要刷新窗口句柄
  CONST RECT *lprcUpdate,     // 要刷新的矩形指针，当 hrgnUpdate 参数有值时无效
  HRGN hrgnUpdate,            // 要刷新的区域句柄
  UINT flags                  // 重画标志
 );
```

hWnd 参数传递 0 表示桌面窗口句柄，flags 参数应传递 RDW_ALLCHILDREN，保证所有包含指定区域的窗口重画。

6.6.2 逐帧动画

除了精灵动画之外，帧动画是另一类动画实现方法，各种格式的动画文件都采用了帧动画。总体来说，基于帧的动画在编程方面不会有太大的难点。

将一组画面连续播放，就形成了帧动画，利用这种方法可以生成动画图标、动画按钮等颇为有趣的效果。

1. 动画图标

经常见到某个应用程序的图标是动态的，例如著名的超级解霸 5.0，这种图标的动画效果是定时改变应用程序图标的结果。

例如，在 OnTimer 事件响应函数中加入：

```
i=(i+1)%6;
Application->Icon->LoadFromFile("bat"+IntToStr(i)+".ico");
```

或是

```
Icon->LoadFromFile("bat"+IntToStr(i)+".ico");
```

都可以达到动画图标的效果。

在配套光盘的 Chapter7\Ex0703\目录下有一个简单的动画图标的例子。这个例子同时演示了动画光标的实现。

2. 动画按钮

动画按钮与动画图标的实现方式相同，例如有一个 BitBtn 按键和一组连续的位图 Earth1~Earth8，在 OnTimer 事件的响应函数中加入：

```
i=(i+1)%8;
BitBtn->Glyph->LoadFromFile("Earth"+IntToStr(i+1)+".bmp");
```

就实现了一个不断变化的按钮。

3. 动画光标和 AVI 动画

Windows 提供一种彩色动画光标格式 ANI（扩展名为".ani"），如何使彩色光标出现在应用程序中呢？LoadCursor 是不行的，解决的方法是使用 LoadCursorFromFile 函数。

```
Screen->Cursors[2]=LoadCursorFromFile("Tifa1.ani");
Form->Cursor=2;
```

AVI 是 Windows 的标准动画格式，以".avi"为扩展名。在 C++ Builder 中提供 TAnimate 组件用于方便的播放这些动画。关于 TAnimate 组件请参看下一章的具体介绍。

6.6.3 多媒体定时器

在 Windows 动画编程中，启用 Timer 定时器播放帧往往是不能达到要求的。因为 WM_TIMER 消息的最短间隔也有 55ms，这将造成不太连贯的图形效果。

对于简短的一段演示性动画来说，可以让动画在一个循环体中不断执行直至播放结束，这样处理的帧与帧之间完全没有时间间隔。但如果不作特殊处理的话，这种做法是不可行的，因为在一段代码中长时间的操作将使应用程序失去处理消息的能力。例如，如果屏保程序采用在循环体中播放帧的方法，屏保将永远不可能退出，其结果与死机无异。

解决这个问题最简单的办法是在循环体中加入 ProcessMesage 方法的调用，在本书的第 17 章中将看到这种方法的应用，它将使动画速度大大提高同时并不失去处理事件的能力。另外

一种差不多的方法是在 Application 类的 OnIdle 事件进行绘图，这个事件在进程空闲时发生，也就是说，只要应用程序不处理消息时就进行绘图。这种方法同样非常有效。

但有时要求必须控制动画的速度，使用 WM_TIMER 消息定时精度又不够，这时可以考虑使用 Windows 多媒体定时器。

1. 多媒体定时器

多媒体定时器是为实现 MIDI 序列播放而设计的，它的精度远远比系统定时器高的多，达到最小 1ms、最大 16ms 的定时器精度，并提供高精度时间片调度和时间中断事件。多媒体定时器可以实现一次定时和周期定时，它与系统定时器有本质不同。系统定时器是将 WM_TIMER 消息发给一个窗口过程，而多媒体定时器则采用中断完成定时服务。并且多媒体定时器是在自己的线程中执行，当时间事件发生后，执行指定的回调函数。

在 mmsystem.h 头文件中包括 2 个多媒体定时器结构的定义和 7 个多媒体定时器函数的声明。与多媒体定时器相关的函数包括下面几个：

- timeBeginPeriod 函数。

设置应用程序或驱动程序使用的定时器最小精度。

- timeEndPeriod 函数。

清除 timeBeginPeriod 函数设置的最小精度。

- timeGetDevCaps 函数。

查询定时器设备以确定其性能，即定时器最小和最大精度。

- timeGetSystemTime 函数。

返回自 Windows 95/98 启动开始经过的毫秒数，其结果存于 MMTIME 结构中。

- timeGetTime 函数。

与 timeGetSystemTime 函数功能相同，但直接返回结果，所以运行开销较小。

- timeSetEvent 函数。

声明：MMRESULT timeSetEvent(UINT uDelay, UINT uResolution, LPTIMECALLBACK lpTimeProc, DWORD dwUser, UINT fuEvent);

设置一个定时回调事件，该事件可以是一次性事件或周期事件，同时通过 lpTimeProc 参数为该事件指定回调函数。

- timeKillEvent 函数。

删除使用 timeSetEvent 函数创建的多媒体定时器事件，释放系统资源。

2. 多媒体定时器编程

使用上述多媒体定时器函数可以创建定时器事件，实现高精度定时控制服务。下面介绍使用多媒体定时器的一般编程步骤：

(1) 确定最大和最小定时器精度。

首先调用 timeGetDevCaps 函数确定多媒体定时器服务提供的最大和最小定时器精度。不同的计算机和不同的 Windows 运行方式，可能具有不同的精度。

(2) 设置最小定时器精度。

在启动定时器服务之前，应用程序必须调用 timeBeginPeriod 函数设置最小定时器精度。

在定时器服务结束后，还要使用 timeEndPeriod 函数清除该值。一个应用程序允许多次调用 timeBeginPeriod 函数设置精度，只需每个调用都与 timeEndPeriod 函数调用相对应即可。

(3) 编写定时器回调函数。

创建定时器回调函数，在该函数中加入定时事件响应时所需要执行的代码。回调函数语法为：

```
void CALLBACK TimeProc(
    UINT uID,          // 标识定时器事件，该标识符是 timeSetEvent 返回的标识符
    UINT uMsg,         // 保留
    DWORD dwUser,      // 提供 timeSetEvent 函数的 dwUser 参数用户实例数据
    DWORD dw1,         // 保留
    DWORD dw2          // 保留
  );
```

(4) 创建定时器事件。

调用 timeSetEvent 函数初始化和创建定时器事件，lpTimeProc 参数指定回调函数地址。

(5) 释放定时器事件。

当应用程序用完多媒体定时器服务后，必须调用 timeKillEvent 函数删除定时器事件。



多媒体定时器函数 timeGetTime 和 timeGetSystemTime 可以单独使用，用于获取非常精确的系统当前时间。

6.6.4 高级动画

本章一直在讨论如何使用 C++ Builder 和标准 Windows 服务来实现动画，这样做的结果是不可能充分的发挥硬件的机能。同时，我们并未涉及到三维动画的话题，因为自己来设计三维动画算法是困难的，也是不合理的。这些问题迫使我们寻求专业的图形 API 来实现高级性能。

DirectDraw（DirectX 的组件之一）和 OpenGL 是两个极为热门同时也是极为强大的图形 API 库。DirectDraw 提供一种类似 DOS 中直接写屏的技术，在二维动画方面大有用武之地，同时 Direct3D 的强大三维图形处理能力和高兼容性也毫不逊色于任何 3D 图形 API。OpenGL 用于专业的三维图像和三维动画编程，它可以利用专为三维显示而设计的硬件性能。OpenGL 建立了一种图形流水线，可以在其中将命令发给图形硬件。

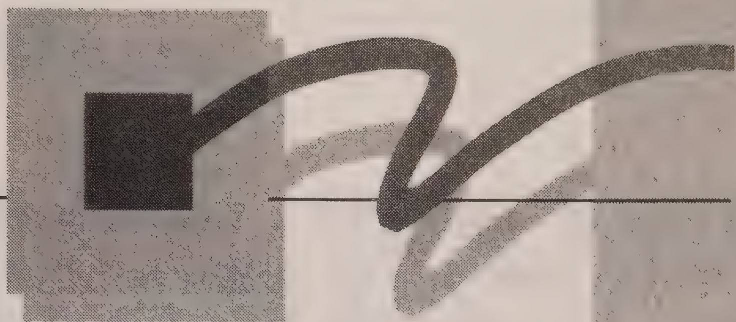
关于如何使用 DirectX 技术，包括使用 DirectDraw 和 Direct3D 编程，我们将在本书第 17 章作具体讨论。

第7章

完美的多媒体播放器

——深入多媒体技术

多媒体技术及编程方法
TMediaPlayer 和 TAnimate 组件
实现多功能媒体播放器程序
应用 MCI 编程
底层多媒体 API



本章导读

在这一章，读者将接触到一些相当有趣的材料，并实现一个漂亮的程序——BCBPlayer 多媒体播放器。该程序能够支持 Wave 音频、MIDI 序列、AVI 动画和 MPEG 影音文件（VCD）等多种媒体格式，提供整套媒体播放控制，音量控制，并拥有一个专业化的播放器界面，同时提供视频播放时的抓图功能等辅助功能。本章将阐述如何利用 C++ Builder 环境中嵌入的多媒体组件和 Windows 多媒体 API 来实现这一切。

另外，本章拓展部分还将介绍应用 MCI 和底层多媒体 API 编程，该部分内容具有一定难度。

本章内容包括:

- 关于多媒体技术、多媒体编程方法的一般讨论。
- 系统介绍 C++ Builder 中两个重要的多媒体组件——TMediaPlayer 和 TAnimate。
- 具体实现多媒体播放器程序 BCBPlayer。结合应用程序的开发，阐述 Borland C++ Builder 中多媒体编程的各个方面的内容和技巧，以及开发过程中涉及的其他技术。
- 问题的深入——应用 MCI 及底层 API 实现高级多媒体性能。

7.1　多媒体技术探秘

在过去的几年，多媒体技术凭借 Windows 的辉煌而飞速发展，今天已是平凡的事物了。生存在 Windows 95/98 的年代，多媒体技术几乎无处不在。广义的范围下，“多媒体（MultiMedia）”这一概念，包括了几乎所有用于计算机的动画、声音和影像。

Windows 95/98 中，从内核到界面和附件都为高性能的多媒体应用程序提供支持。Windows 95/98 为编程人员提供了两种主要界面用于打开和操纵多媒体设备，包括媒体控制接口 MCI（Media Control Interface）和多媒体相关的底层 API。

本节将对多媒体设备和多媒体技术加以具体讨论。

7.1.1　多媒体技术的核心

通俗的讲，多媒体（Multimedia）技术是以计算机为基础，以文字（Text）、声音（Audio）、图片（Picture）和图像（Image）等媒体作为信息来源，形成一种人机交流的媒体。通过这些媒体，用户能以自然感觉的方式处理并使用信息，使信息表现为图、文、声、像并茂，这样就可以使计算机更加人性化，使一般使用者更易于了解和使用计算机。在多媒体计算机中，用户可以通过鼠标、甚至声音向计算机发送控制信息，计算机则通过图形界面、音频、视频等更加直观的方式来反映应用程序的运行情况。

多媒体技术最终要处理的无非是图、文、声、像四类信息，并解决这些信息的表示、传送、存储和处理问题。作为一个程序员来说，图、文、声、像四类信息的表示和处理问题是我们所需要关注的多媒体技术的核心。

可能与所想象的有略微不同，在多媒体技术领域，多媒体信息的传输和存储占有更加重要的地位。我们知道，在计算机中信息是以二进制数存储，信息存储的最小单位是位。对于文字信息来说，一个英文字占用一个字节（8 位），一个汉字占用一个字（16 位），文字信息占用空间很少，因而存储和传输不成问题。然而，视频和音频信息都有极大的数据量。以一幅每秒 30 帧 640×480×24 位色的动态图像为例，每秒的信息量就有近 30MB，音频信息也大致如此。所以，多媒体技术的发展是与信息的传输和存储，包括信息的压缩和解压技术紧密相连的。

7.1.2　Windows 操作系统的多媒体服务

要实现多媒体系统，除了需要构成多媒体设备的硬件之外，还必须具备控制多媒体系统的软件。实际上，Mircosoft 在其 Windows 3.x 操作系统以及后来的 Windows 95/98，Windows

NT 操作系统均提供了面向多媒体设备的服务和支持。为满足不同层次的用户，这些控制服务分为高层服务和底层服务。包括有控制媒体设备的媒体控制接口 MCI（Media Control Interface）以及支持多媒体相关服务的底层 API。

1. 媒体控制接口 MCI

Windows 操作系统的多媒体控制接口（MCI）是一个与设备无关的接口。它已包含在 Windows 的公共动态链接库文件 mmsystem.dll 的 mmsystem 模块中，用于协调多媒体软件与 MCI 设备驱动程序之间的通信。通过此接口有三种方式控制多媒体设备，包括：

- 通过 MCI 接口直接控制设备，如 CD 设备等。
- 通过使用 mmsystem 来间接控制媒体设备。
- 另外一些 MCI 设备驱动程序，如 MCI 影像设备等，提供了与其他 Windows DLL 的高级接口。

图 7-1 表示多媒体应用程序与多媒体驱动程序之间的交互关系，应用程序与 MCI 设备之间通过 mmsystem 模块隔离，从而确保应用程序与设备之间的无关性，这正是采用 MCI 接口开发多媒体应用程序的优点所在。

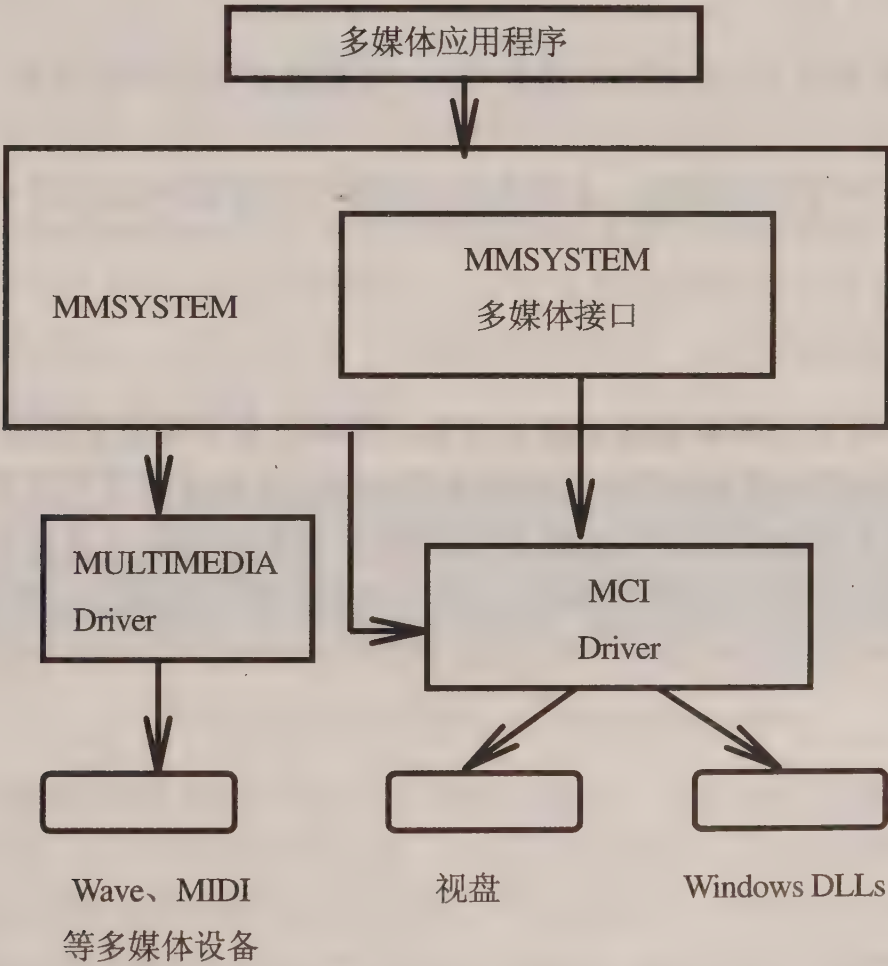


图 7-1　多媒体应用程序与多媒体驱动程序之间的交互关系

2. MCI 编程接口

MCI 提供两种类型的编程接口——命令串（Command-String）接口和命令消息（Command-Message）接口。命令串接口是通过 ASCII 字符串来发送驱动程序的命令，命令消息接口是通过消息和数据结构为多媒体设备发送命令，并接受由设备传送的信息。

例如命令串接口的主要函数 mciSendString，可以向多媒体设备发送字符串形式的命令。mciSendString 函数定义在 mmsystem.h 中，其原型如下：

```
MCIERROR mciSendString(
  LPCTSTR lpszCommand,
  LPTSTR lpszReturnString,
  UINT cchReturn,
  HANDLE hwndCallback
 );
```

其中第 1 个参数 lpstrCommand 用以传送 MCI 命令的字符串，例如下面的一段程序：

```
#include <mmsystem.h>
#include <string.h>
void main()
{
     char buff[50];
     mciSendString("open cdaudio",buff,strlen(buff),NULL);
}
```

此程序利用 MCI 函数打开 CD-ROM 播放 CD。如果播放操作出现异常，错误信息将存入 buff 中。

MCI 的功能是非常丰富和有效的。本章将在实现高性能多媒体部分进一步讨论应用 MCI 的多媒体编程。

3. MCI 设备

MCI 设备驱动程序可分为简单设备和组合设备。简单设备不需要重放的数据文件，如 CD 唱盘等；组合设备需要重放的数据文件，如 MIDI 序列发生器和音频波形发生器。

可以察看控制面板的多媒体项中的设备页来了解计算机附带的多媒体设备及其相应的驱动程序。而一个更加明了的方法是察看 System.ini 文件的[MCI]字段，情况可能如下所示：

```
[MCI]
cdaudio=mcicda.drv
sequencer=mciseq.drv
waveaudio=mciwave.drv
avivideo=mciavi.drv
videodisc=mcipionr.drv
vcr=mcivisca.drv
MPEGVideo=mciqtz.drv
```

7.1.3 C++ Builder 的多媒体编程

C++ Builder 通过 TMediaPlayer 组件提供多媒体编程支持。如前所述，MCI（Media

Control Interface）接口是 Mircosoft 公司为实现 Windows 下与设备无关的媒体控制而提供的接口标准。通过 MCI 指令，用户可以直接对多媒体外部设备实施控制，或从媒体设备上获得相关信息。Windows 系统提供了 100 多个具有多媒体处理能力的 API 函数。通过函数调用实现多媒体编程是复杂的，而 C++ Builder 中提供的 TMediaPlayer 组件封装了 MCI 中的大部分功能，因此使用 TMediaPlayer 组件将使编程人员能够轻松自如的实现多媒体调用。TMediaPlayer 组件主要支持 CD 音频，Wave 音频文件，MIDI 序列文件和 AVI 动画文件等多媒体应用。

CD 音频文件为在多媒体计算机上用 CD-ROM 驱动器从 CD 唱片上播放 Red Book 音频信息提供支持。CD 音频文件提供最高的音乐再现质量，但相对来说它所占的存储空间也最大——176KB/s 的音频传输速率。

Wave 文件是 Mircosoft 的标准文件格式，通常用以读取非音乐音响，如说话声音等等。Wave 文件使用 11KB 的磁盘空间存储 1s 的声音，或用 1MB 的空间存储 1min 的声音。这种文件通常占据非常大的磁盘空间，所以它常用于增加短促的声响效果。

MIDI（Musical Instrument Digital Interface）是乐器数字接口文件。简单的说，MIDI 文件可以被认为是包含了一系列的音符，如升 C 或降 A 等。这些音符和指令一起被传送给音序合成器，通过模拟乐器合成这些音符。MIDI 文件存储 1min 的音乐需要 50KB 左右的磁盘空间（但可能因为同时演奏的乐器的多少有很大的不同）。MIDI 文件占用空间约为 Wave 音频文件的 1/20，所以在一些方面 MIDI 文件很有用，它可以为应用程序提供前奏和背景音乐。

AVI（Audio Video Interleave）文件是一种特殊的 RIFF（资源交换文件）文件。RIFF 文件允许程序以灵活的可扩展的方式存储数据，它允许复杂的数据结构被构造为能够快速访问的、没有存储局限的形式。AVI 文件基于快速访问的要求，是没有经过压缩的文件格式。它通常也较大。TMediaPlayer 组件如图 7-2 所示。

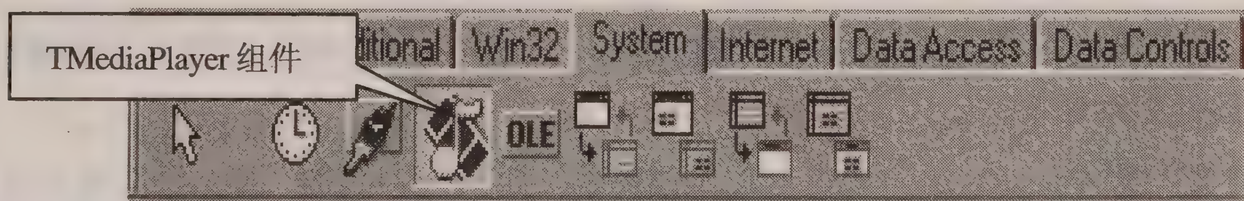


图 7-2　System 选项板上的 TMediaPlayer 组件

C++ Builder 创造性的将 MCI 中的 API 函数封装在一个可视化的播放器组件中。所以，在 C++ Builder 中尽可以写一两行代码就完成一个音乐播放器。应用这样一个出色的组件在程序中实现各种类型的多媒体编程都是相当轻松的。TMediaPlayer 组件显示一个 VCR 风格的控制面板，可以通过设定属性来控制该组件上的各个按钮，如改变显示颜色、增减按钮数目等。当然，也可以把 TMediaPlayer 组件的 Visible 属性设为 false，放弃这个播放控制面板，而由程序控制多媒体文件的播放控制。

C++ Builder 的多媒体播放器组件是对 MCI 功能的一个成功的封装，但是它并没有封装 MCI 的全部功能（虽然在绝大多数的情况下，TMediaPlayer 组件已足够用）。所以在特定的情况下，仍然需要直接依靠 MCI 来控制多媒体操作。甚至在要求更高的情况下，我们还不得不使用 Windows 底层多媒体 API 实现难度更大的多媒体编程任务。

7.2　多媒体相关组件和多媒体编程

C++ Builder 中与多媒体有关的组件包括图像组件 TImage、媒体播放机组件 TMediaPlayer 和动画组件 TAminate。有关图像显示处理技巧和图像组件 TImage 的应用在此前已有详尽的阐述，这里将简单讨论 TAminate 组件和 TMediaPlayer 组件。

7.2.1　多媒体 TMediaPlayer 组件

媒体播放机 TMediaPlayer 组件是 C++ Builder 多媒体控制和多媒体应用的主要手段。它功能强大且极易使用，对 CD 播放、AVI 影片播放、Fic、Fli 动画文件、Midi 音乐文件播放与 Wave 声音播放文件录入等功能全部都可以轻易地实现。TMediaPlayer 位于组件选项板的 System 页，它本身包括一组按钮（如 Play、Stop 等，如图 7-3 所示）。



图 7-3　TMediaPlayer 组件放在一个标准窗体中间

1. TMediaPlayer 组件的重要属性

TMediaPlayer 组件是一个功能强大的多媒体组件，提供一组与 MCI 功能对应的属性，其中比较重要的属性列举如下。

- AutoOpen 属性。

自动打开设备。决定在程序执行时，使某一媒体设备自动处于打开状态。如果已把此属性设为 true，但 FileName 属性值为空，会造成出错信息，属性返回布尔值。

- AutoRewind 属性。

设置媒体播放机是否自动复位。

- FileName 属性。

文件名字段，用于存储或打开的文件名。值得注意的是，如媒体类型为 CDAudio 时，FileName 属性的值必须为空串，否则就会出现错误。

- DeviceType 属性。

指定要打开设备的类型。在这里面还有许多选项，主要指明 MCI 的各类设备，其可用值为：

dtaviVideo	AVI 动画设备。
dtDAT	数字音频磁带机。
dtOther	其他未定义的 MCI 设备。

dtWaveAudio	数字化波形文件。
dtScanner	图像扫描仪。
dtSepquence	MiDi 音讯器。
dtVCR	盒式录像机。
dtOverlay	在小窗口中播放模拟电视。
dtMMMovie	MMM 动画文件。
dtAutoSelect	根据 FileName 中所指定的文件类型自动对应媒体设备，并取决于当前系统所安装的媒体类型。

- Shareable 属性。

是否与其他程序共用一个设备。

- Wait 属性。

等待命令执行完以后,再执行下一条命令。

- Notify 属性。

决定下一个命令是否使用 MCI 通知服务。

- Capabilities 属性。

显示开启媒体设备拥有的功能。为只读集合属性，集合元素值为 mpCanStep、mpCanEject、mpCanPlay、mpCanRecord 和 mpUsesWindow。

- TrackLength 属性。

每一轨道的时间长度。

- TrackPosition 属性。

每一轨的起点时间位置。

- Tracks 属性。

目前媒体所在轨数。

- DisplayRect 属性。

可以调整显示区域位于该控制控件的位置。注意：此属性仅在媒体设备打开时才能被设定。

- Display 属性。

决定视频文件（如 AVI，VideoCD）播放时的显示窗口。如 Display 属性为空，则自动生成一个窗口用于显示输出，从 TWinControl 继承的组件都可以作为媒体视频的输出窗口，如 TForm、TPanel 等等。例如:

```
MediaPlayer1->Display=ShowForm;
MediaPlayer2->Display=ShowPanel;
```

- Mode 属性。

决定当前 TMediaPlayer 的 MCI 模式。

- Position 属性。

决定媒体目前的时间位置，返回值为长整形。

- Frames 属性。

Frames 属性决定媒体播放组件的 Step 方法前进和后退操作所移动的帧数。

- ErrorMessage 属性。

字符串类型，对操作所产生的错误进行解释。

2. TMdiaPlayer 组件的重要方法和事件

C++ Builder 的多媒体播放机组件 TMediaPlayer 除了提供与组件按钮能够相对应的方法 Play、Pause、Stop、Next、Prve、Step、 Record、Eject 等之外，还提供了其他的一些有用的方法。包括：

Resume　　　继上一暂停位置播放。

StartRecording 开始录音。

Rewind　　　复位。

Save　　　　存储媒体到文件。

TMediaPlayer 只有几个事件，但对于媒体控制非常有用。

- OnNotify 事件。

当媒体控制方法完成时，OnNotify 事件被激活。OnNotify 事件与 TMediaPlayer 的 Notify 属性相关联——为了使 OnNotify 事件能够发生，必须在控制方法完成前置为 true，而为了使下一次的 OnNotify 事件能够发生，Notify 属性须再次重置为 true。OnNotify 事件可以使用户在媒体播放机的动作完成时得到通知，是使用 MediaPlayer 时的一极为重要的事件。

- OnPostClick 事件。

当 OnClick 事件处理程序代码被调用后，OnPostClick 事件被触发。如媒体播放机的按钮被单击，同时 Wait 属性被置为 true，则直到 OnClick 代码完成后，OnPostClick 事件才被触发。而如 Wait 属性值被设置为 false，那么在 OnClick 程序代码完成之前，控制就被交给了应用程序，并且完全可能在 OnClick 事件完成前触发 OnPostClick 事件。

3. CD 播放编程

作为使用 TMetiaPlayer 编写多媒体程序的例子，我们将实现一个类似于 Windows CD 播放器的应用程序，其运行结果如图 7-4 所示。CD 播放支持在本章的主要范例——BCB 多媒体播放器中将不再实现。



图 7-4　简单的 CD 播放器示例

利用 C++ Builder 实现 CD 播放功能是容易的。但播放 CD 编程需要注意一些问题。首先，播放 CD 虽然没有媒体文件，但仍然需要用 Open 方法打开，并且 FileName 字段应该置为空字符串；其次，我们在程序中应该判断当前光驱是否有 CD 唱片和 CD 唱片是否可以播放。例如本例中 CD 播放的初始化代码如下：

```
void __fastcall TForm1::SpeedButton1Click(TObject *Sender)
{
    MediaPlayer1->FileName="";
```

```
        MediaPlayer1->Wait=true;
        MediaPlayer1->DeviceType=dtCDAudio;
        MediaPlayer1->TimeFormat=tfTMSF;
        MediaPlayer1->Notify=true;
        isOpen=true;
        try{
            MediaPlayer1->Open();
        }
        catch(...){
            Application->MessageBox("光驱中没有可用的 CD 唱盘",
                "错误",MB_OK+MB_DEFBUTTON1+MB_ICONSTOP);
            isOpen=false;
        }
        Timer1->Enabled=false;
    }
```

在这个 CD 播放器示例程序中，实现播放进度时间的显示是一个较为关键的问题。它的实现方式和本章实例 BCB 多媒体播放器中的实现方式相同，在本章稍后会做讲解。

7.2.2 动画组件 TAnimate

C++ Builder 中提供的动画组件 TAnimate 是 Win32 动画控件（Animation Controls）的一个可视化封装，TAnimate 组件位于 C++ Builder 组件栏的 Win32 选项板上，如图 7-5 所示。

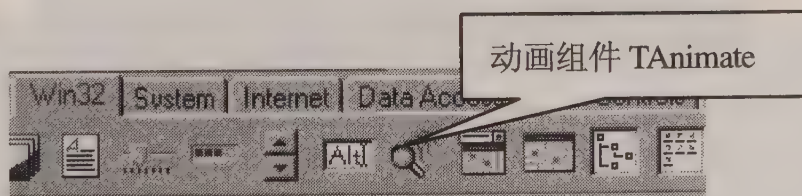


图 7-5　C++ Builder 中的动画组件

TAnimate 组件主要应用于简单形式的动画播放，TAnimate 组件只支持 Audio Video Interleaved Clip，比 TMediaPlayer 播放 AVI 文件显得更简洁和有效率，但 TAnimate 并不支持 AVI 文件中的声音。

以下简单阐述 TAnimate 组件的应用方法和应用中应注意的问题。

1. TAnimate 组件的重要属性

- FileName 属性。

返回或设定播放的 AVI 剪辑的名称。

- Active 属性。

返回布尔值，通过 Active 属性可以判明是否正在播放 AVI 动画文件。

- Repetitions 属性。

确定 AVI 动画文件重复播放的次数。

- CommonAVI 属性。

可以从 Shell32.dll 中提取系统动画来播放。其值包括 aviFindFolder、aviFindFile、aviFindComputer、aviCopyFiles、aviCopyFile、aviRecycleFile、aviEmptyRecycle 和 aviDeleteFile。使用系统动画时，FileName 字段应置为空。

TAnimate 组件的方法有 Play，Seek 等。Play 方法的声明为：

```
void __fastcall Play(Word FromFrame, Word ToFrame, int Count);
```

FromFrame	播放的起始帧，可以置为 StartFrame 常量，置为 1 则表示从文件的物理起始帧开始播放。
ToFrame	播放的终止帧。可以置为 StopFrame 常量，置为 0 表示播放到文件末尾。
Count	定义播放次数。

Seek 方法用于播放所指定的帧，Seek 方法的声明为：

```
void __fastcall Play(int Frame);
```

Frame 参数用于设定播放的帧序号。

在 Windows 95/98 中很多地方都用到了动画控制，例如文件拷贝时，动画剪辑形象地表明了文件复制的全过程。

利用 C++ Builder 中的动画组件，我们可以更加容易的实现这一功能。图 7-6 即是仿 Windows 95 拷贝的演示程序。



图 7-6　使用 TAnimate 组件的复制演示程序

在 C++ Builder 中进行多媒体编程非常之简单。这一切都得归功于 Inprise（原 Borland 公司）公司天才的程序员们，他们出色封装了 Win32 的多媒体功能。这种情况可能使得在将来，初学者的第一个程序不再是经典“Hello World!”，而是只写一两行代码就可完成的多媒体程序。

7.2.3　多媒体编程的一般原则

一般来说，以 Windows 为平台开发多媒体应用程序应该遵循以下原则：

(1)　多线程限制。

Windows 95/98 的多媒体函数不能被同一进程内的两个或多个线程同时调用。虽然对大多数多媒体函数而言，被多个线程调用也能正确工作，但极容易失败。

(2)　硬件兼容性。

在 Windows 系统中，多媒体 DLL 存在并不能保证有合适的设备驱动程序可用。

(3) 空间与时效。

多媒体给我们带来了华丽的影像和动听的音效，但这些动人的效果是建立在极大信息量基础上的。一般来说，不经数据压缩的媒体文件格式（如 Bitmap、AVI、Wave 等）的再现速度并不存在问题，但这类媒体文件要占用大量的磁盘空间，所以这类文件一般用在要求速度的场合；另一类文件经过了压缩（MPEG、JPG）或为特殊纪录格式（MIDI），经过压缩使得空间大大减小，但为此却牺牲了速度。

在多媒体程序中，使用哪种格式的媒体文件是应当仔细考虑的问题。

(4) 让出资源。

长期占用设备的应用程序应在其处于非活动状态期间，不再显示动画或插播声音，为其他程序让出多媒体设备资源，以使其他程序可以正常运行。但有一个例外，对于播放 CD 音乐的那些应用程序，因为用户可能需要在当另一程序运行时继续播放 CD。若另外有个应用程序需要使用 CD 驱动器，程序必须告诉用户发生了冲突。

应用程序应提供关掉某些多媒体特性的选项，特别是声音。这是因为，用户可能希望在他们运行某个应用时有一个后台，用来播放多种媒体音乐，并对音乐播放进行相应的控制，包括在不希望播放时关掉音乐。

7.3 实例创建分析

C++ Builder 的媒体播放组件确实强大，但我们显然不会满足那个并不好看的播放面板存在于程序界面中，可能会更加青睐于那些专业的音乐播放器。实际上，我们完全可以利用 C++ Builder 的媒体播放组件，轻松开发一个与专业音乐播放器同样出色的多媒体播放器 BCBPlayer，如图 7-7 所示。



图 7-7 多媒体播放器 BCBPlayer

BCBPlayer 是一个界面华丽、并支持多种格式的多媒体播放器。该播放器具有下面几个特点：

- 对多种格式的音频、视频文件都有很好的支持。
- 具有良好的应用程序界面。
- 实现强大的辅助功能，如音量调整、播放视频文件时的抓图功能等。

开发 BCBPlayer 程序需要在应用程序中实现多媒体功能，但我们并不想显示媒体播放组件的外观（对于有较高审美要求的程序来说，媒体播放组件的播放面板实在不能让人满意）。这是容易办到的，只需把媒体播放机组件的 Visible 属性设为 false，而改用在程序中调用 MediaPlayer 对象的相应方法来控制媒体播放。这是我们实现 BCBPlayer 程序的基本考虑。

开发 BCBPlayer 媒体播放器程序首先要做的是界面设计和制作。程序的界面部分可分为四大块：主窗体和背景、信息显示板部分、控制和功能按钮、装饰图像和非可视组件。界面

的制作完全在 C++ Builder 集成开发环境中完成。

BCBPlayer 应用程序实现的大致框架为：

- 基本的媒体控制部分。包括播放器面板上几个控制按钮（播放、停止、打开等）的响应处理部分。其中媒体文件的打开处理是问题的重点；而后应完成媒体播放状态的响应处理部分，主要是编写 TMediaPlayer 的 OnNotify 事件的响应函数，以及相关处理。
- 视频文件播放的处理部分。这里我们建立新的用于显示视频的窗体。然后在主窗体部分和视频显示窗体部分加入相应代码，解决视频播放的问题。
- 进度时间显示的实现部分。这涉及到 C++ Builder 中资源文件的使用，以及播放进度时间的计算和显示。
- 必要的辅助功能的加入。包括视频播放时控制面板的弹出功能、视频播放时的抓图功能、控制面板的拖动功能、音量调整功能。

BCBPlayer 工程中主要包括以下文件：

- 播放器窗体单元。类名：TForm1，文件：Unit1.h、Unit1.cpp。
- 视频输出窗体单元。类名：TForm2，文件：Unit2.h、Unit2.cpp。
- 音量调整窗体单元。类名：TForm4，文件：Unit4.h、Unit4.cpp。
- 资源文件 Num.res。包含了用于显示时间的数字位图。

7.4 媒体播放部分的实现

本节开始着手实现媒体文件播放和控制，包括处理音频、视频播放的一些问题。利用 C++ Builder 的 TMediaPlayer 组件实现这一切并不困难，但应该注意对一些细节进行周到的考虑。基本媒体播放控制在于三个方面：媒体文件导入处理、功能键响应处理和媒体播放状态响应。

7.4.1 基本媒体播放控制

1. 媒体文件导入处理

应用 TMediaPlayer 组件播放媒体文件时，首先要通过 Open 方法来打开媒体文件。响应播放面板上的打开按钮，函数 SpeedButton5Click 进行有关媒体文件导入的处理工作。

打开按钮处理函数——SpeedButton5Click 函数完成的主要工作如下：

首先判断 MCI 设备是否已被使用；若被使用，则先关闭 MCI 设备；而后设置 DeviceType 为 dtAutoSelect，用以自动支持各种类型媒体文件；再设置 OpenDialog 对话框的打开文件类型。

```
if(MediaPlayer1->Mode==mpPlaying||MediaPlayer1->Mode==mpOpen)
    MediaPlayer1->Stop();
    MediaPlayer1->DeviceType=dtAutoSelect;
```

```
OpenDialog1->Filter="All files (*.*)|*.*|Wave files (*.wav)|*.Wav|MPEG files (*.dat)|*.Dat|AVI files (*.avi)|*.Avi|MIDI files (*.mid)|*.Mid";
```

其后我们就可以激活 OpenDialog 对话框，进行有关媒体文件导入的处理。

```
if(OpenDialog1->Execute())
{
	MediaPlayer1->FileName=OpenDialog1->FileName;
	MediaPlayer1->Open();
	// 在显示面板上显示媒体文件名
	Label3->Caption=ExtractFileName(MediaPlayer1->FileName);
	// 初始化进度时间，并显示之。
	mMinutes=0;
	mSeconds=0;
	DisplayTime(0,0,0);
	// 判断是否是视频文件，若是，则进行有关视频文件的处理。
	// 关于视频部分处理问题稍后将有具体讨论。
	if (ExtractFileExt(MediaPlayer1->FileName).LowerCase()==".dat"||
		ExtractFileExt(MediaPlayer1->FileName).LowerCase()==".avi"){
		Form2->Show();
		MediaPlayer1->Display=Form2;
		Form2->Caption=ExtractFileName(MediaPlayer1->FileName);
		MediaPlayer1->DisplayRect=Rect(0,0,Form2->ClientWidth,Form2->ClientHeight);
		isVideo=true;
	}
	else if (isVideo) {
		Form2->Close();
		isVideo=false;
	}
	// 因为没有开始播放，将用于计算和处理进度时间显示的定时器 Timer1 关掉。
	Timer1->Enabled=false;
}
```

2. 功能键响应

在设计程序界面时，我们制作了几只美观的 SpeedButton 按钮，包括播放、暂停、停止等。程序中需要完成相应的响应程序代码。

这并不是一件复杂的工作。但是，我们应该对问题进行缜密的思考，保证程序的健壮性和完善性。

响应“播放”按钮函数 SpeedButton2Click 中，播放并应保证定时器组件打开（即 Enable 属性为 true），时间显示开始。

```
void __fastcall TForm1::SpeedButton2Click(TObject *Sender)
{
```

```
        MediaPlayer1->Play();
        Timer1->Enabled=true;
    }
```

响应“暂停”按钮函数 SpeedButton3Click 中，播放并应保证计时器关闭。

```
    void __fastcall TForm1::SpeedButton3Click(TObject *Sender)
    {
        MediaPlayer1->Pause();
        Timer1->Enabled=false;
    }
```

响应“停止”按钮函数 SpeedButton4Click 中，播放并应保证计时器关闭；同时播放位置应回到文件开始。

```
    void __fastcall TForm1::SpeedButton4Click(TObject *Sender)
    {
        MediaPlayer1->Stop();
        MediaPlayer1->Position=MediaPlayer1->Start;
        Timer1->Enabled=false;
    }
```

响应“关闭”按钮函数 SpeedButton1Click 中，使程序关闭。注意在关闭的同时响应 Form1 的 OnClose 事件，做一些关闭前的处理工作，如关闭媒体播放组件、释放对象等。

```
    void __fastcall TForm1::SpeedButton1Click(TObject *Sender)
    {
        MediaPlayer1->Close();
            delete numBmp[i];   ……
        Form1->Close();
    }
```

3. 媒体播放状态响应

在前面已经提到，媒体播放状态改变将触发 OnNotify 事件，在 C++ Builder 的多媒体编程中通常需要响应这个事件，根据播放状态改变做相应处理。

在响应函数 MediaPlayer1Notify 中，我们关注的是 MediaPlayer1 的 Mode 和 NotifyValue 属性。Mode 属性和 NotifyValue 属性分别对应于当前 Windows MCI 模式和 Windows MCI 驱动程序消息。

Windows MCI 驱动程序的信息来自以下四个方面：

mci_Notify_Successful	在命令结束时发送。
mci_Notify_Superseded	命令被另一个函数终止。
mci_Notify_Abouted	当前函数终止。
mci_Notify_Failure	发生错误。

但 TMediaPlayer 组件并不直接把这些信息传送给程序，而是转化成为 nvSuccessful、

nvSuperseded、nvAborted 和 nvFailure 四个相应的值存储在 NotifyValue 属性中，我们可以通过访问 NotifyValue 属性获取这些值，并做相应处理。

然而，要真正弄清楚 MCI 设备程序何时发送何种消息并不是件容易的事。作为规则，在以下情况会收到这些消息：

- nvSuccessful 消息是一个文件终止时发出的。
- nvSuperseded 消息可能来自于系统被暂停而发生的，而用户可以随后执行 Play()方法。
- nvAborted 消息可能是 Pause 方法被执行时发出的，另一种可能是将要关闭设备引起的。

当前的 MCI 模式由 TMediaPlayer 的 Mode 属性所决定。MCI 模式常量包括：

```
mci_Mode_Not_Ready
mci_Mode_Stop
mci_Mode_Play
mci_Mode_Record
mci_Mode_Seek
mci_Mode_Pause
mci_Mode_Open
```

每一种模式都有自身的解释。与 MCI 驱动程序消息一致，MCI 模式常量在 C++ Builder 中也并不直接传给用户，而是代以相应的解析量存储在 Mode 属性中，如 mpNotReady、mpStopped、mpPlaying、mpRecording、mpSeeking、mpPaused、mpOpen 等等。

通过 Mode 和 NotifyValue 属性，我们就可以确定播放状态，并完成相应处理。如在 BCBPlayer 程序中的响应函数 MediaPlayer1Notify 编写如下：

```
void __fastcall TForm1::MediaPlayer1Notify(TObject *Sender)
{
    if(MediaPlayer1->Mode==mpStopped||MediaPlayer1->NotifyValue==nvSuccessful){
        Timer1->Enabled=false;
        mMinutes=0;
        mSeconds=0;
        DisplayTime(0,0,0);
    }
    MediaPlayer1->Notify=true;
}
```

如果播放停止（不是暂停）或播放文件结束，应将时间置为 0，并显示之（即执行 DisplayTime(0,0,0)）。最后应该注意的是，在响应函数中应把 Notify 属性置为 true，以使程序能够再次响应 OnNotify 事件。

7.4.2　视频播放相关处理

在视频播放中，我们要处理三方面问题。

1. 处理显示窗体

此前我们提到过 TMediaPlayer 的 Display 属性，这个属性用于指定视频媒体的播放窗口。若 Display 值为 NULL，则 TMediaPlayer 组件将自动生成一个窗口用以显示，但这个窗口没有最大化功能及窗口改变尺寸功能，而且也不能由程序进行控制。

因此，我们需要自己实现一个窗体 Form2，用 MediaPlayer1->Display=Form2 来指定显示窗口。但只是这样还是不够的，因为 MediaPlayer 并不会自动将视频图像填满整个窗口，而只是把视频图像在窗口的“一角”显示出来。我们所需的是在任何时候视频显示区域与 Form2 的大小一致，这时应该关注 MediaPlayer1 的 DisplayRect 属性。

Form2 被显示时，任何时候 DisplayRect 属性值都应与 Form2 的 ClientRect 值保持一致。对于 Form2 窗体，需要处理 OnResize 事件——当用户改变窗体大小或最大化时，响应函数中能够调整 DisplayRect 属性值使之与 Form2 的 ClientRect 值再次一致。

```
void __fastcall TForm2::FormResize(TObject *Sender)
{
        Form1->MediaPlayer1->DisplayRect=
                Rect(0,0,Form2->ClientWidth,Form2->ClientHeight);
}
```

2. 视频播放时控制面板的弹出和消隐

作为一项方便用户的功能，多数专业的 VCD 播放器（如超级解霸、XingPlayer 等）在显示窗体上单击鼠标，控制面板就会弹出，再次单击，面板则会消隐。编程实现这种功能并不困难。

BCBPlayer 程序中定义 isTop 布尔量纪录面板的状态，表示面板是否显示在视频播放窗口之上，并在 Form2 中响应 OnClick 事件，其响应函数如下：

```
void __fastcall TForm2::FormClick(TObject *Sender)
{
        if(isTop){
                Form1->SetFocus();
                isTop=false;
        }
        else isTop=true;
}
```

如果面板已弹出，即控制面板窗体是焦点窗体，这时单击显示窗体，显示窗体成为焦点窗体，则面板被自动消隐；若面板未弹出，即显示窗体是焦点窗体，这时可以用 SetFocus 方法使面板窗体为焦点窗体，面板将弹出并显示于视频窗体上方。

3. 视频抓图功能

在视频显示窗口上设置一个弹出菜单，用户单击菜单上的抓图项即可实现视频播放时的抓图功能。抓图功能实现主要是使用 TCanvas 的 CopyRect 方法。在用户发出抓图命令时，首

先在内存中创建一个位图对象，用 CopyRect 方法将 Form2 上的视频图像拷入位图对象，随后再用 TBitmap 对象的 SaveToFile 方法存入文件。但要注意执行上述步骤前应首先暂停视频文件的播放。

```
void __fastcall TForm2::N1Click(TObject *Sender)
{
    Form1->MediaPlayer1->Pause();
    Graphics::TBitmap *bitmap=new Graphics::TBitmap;
    bitmap->Width=Form2->ClientWidth;
    bitmap->Height=Form2->ClientHeight;
    bitmap->Canvas->CopyRect(Rect(0,0,Form2->ClientWidth,
        Form2->ClientHeight),Form2->Canvas,
        Rect(0,0,Form2->ClientWidth,Form2->ClientHeight));
    if(SaveDialog1->Execute())
        bitmap->SaveToFile(SaveDialog1->FileName);
    Form1->MediaPlayer1->Play();
}
```

应注意的是：这里 SaveDialog1 不应先于 CopyRect 方法执行，虽然理论上没用什么问题。不先执行 SaveDialog1 的理由是，SaveDialog 执行时弹出的对话框会留下一些“残像”在视频显示窗体之上（因 SaveDialog 关闭时，视频显示窗体还未来得及重画），后执行 CopyRect 方法存入的位图会遭到这种“残像”的破坏。



以上的方法看似没有什么问题，但令人惊奇的是，此种方法只适用于 AVI 视频，而对于 MPEG 类型的文件是无效的！这是由于 MPEG 设备类型采用了特殊的显示方式。

7.5 其他关键问题处理

多媒体播放器 BCBPlayer 的基本框架已经完成，但作为一个完整的播放器应用程序，还要进行一些必要的“关键问题”的处理。包括位图数字显示、时间进度的处理和面板拖动的实现。解决这些问题涉及了 C++ Builder 编程中一些非常有用的技巧。

7.5.1 数字显示实现——使用资源文件

为了满足程序审美上的需要，我们采用了一组位图数字显示时间，包括 0 ~ 9 的 10 张位图。为方便程序中位图数字的装入，一个合适的方法是将这组位图放入一个资源文件，同时在 BCBPlayer 程序中，调用 LoadFromResourceName、LoadFromResourceID 方法装入内存。图 7-8 给出了应用位图数字的显示面板。

在上一章中我们已经接触过 C++ Builder 中资源文件（Res 文件）的应用。其实在 C++ Builder 中还可以使用 Win32 的标准资源文件 RC 文件。.RC 文件与.Res 文件略有不同，RC

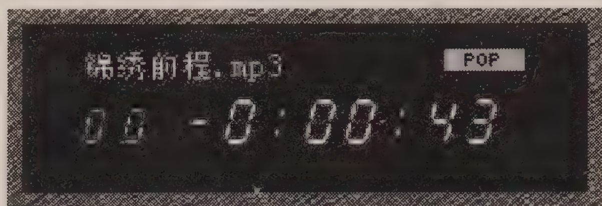


图 7-8 应用位图数字的显示面板

文件仅是一个用于资源标识的纯文本文件，并不包含资源数据，而 Res 文件包含各种资源屏且经过编译。例如，可能有如下的一个 RC 资源文件，它定义了一个声音资源，一个光标资源，一个图标资源，两个位图资源和一个 EXE 文件资源。

src1.rc：

```
Body BITMAP "body1.bmp"
BackGround BITMAP "bg.bmp"
BgSound WAV "bgsound.wav"
Cursor1 CURSOR "cursor1.cur"
Icon ICON "file1.ico"
File EXEFILE "xcopy.exe"
```

Tip

在 C++ Builder 程序中，我们可以直接使用.RC 文件。例如一个.RC 文件中包含了位图资源，可以使用 LoadBitmap API 函数调入内存中。一些其他种类的资源也有相应的访问函数，包括：图标 LoadIcon、光标 LoadCursor、加速表 LoadAccelerators、菜单 LoadMenu、字符串 LoadString。而对于像声音、动画等不能直接访问的资源，可以通过依次调用 FindResource、LoadResource、LockResource 三个 API 函数来使用这个资源。

C++ Builder 中，我们更愿意使用.RES 资源文件。实际上，可以先写一个.RC 文件（RC 文件的制作相比之下更加灵活和容易），然后用 C++ Builder 所带的 brcc32.exe 工具将 RC 文件和相应资源合编成一个 RES 资源文件。

Brcc32 工具用法如下：

格式： BRCC32 [选项 ...] 文件名

主要选项含义：

-r	忽略兼容性。
-32	编译为 32 位系统兼容的 .RES 文件，为默认值。
-m	能够支持多字节字符。

在本章范例程序 BCBPlayer32 中，也包含了一个名为 Num.res 的资源文件。采用的上述方法：首先完成一个 Num.rc 文件，内容如下：

```
Num0 BITMAP 00.bmp
Num1 BITMAP 11.bmp
......
```

然后，编译成 Num.res 文件，加入工程中，并使用#pragma reaource "num.res"预处理命

令。至此程序中就可以用 LoadFromResourceName 装入位图，从而实现面板上的数字显示：

```
tmpBitmap-> LoadFromResourceName(0,"NUM1");
```

7.5.2 播放时间进度显示

对于多媒体播放机程序，显示播放的时间以标明播放的进度是必要的。和媒体文件播放进度相关的是 TMediaPlayer 组件的 Position 属性，Position 属性根据媒体不同的时间格式而得到 4 字节的值。但时间格式（TimeFormat 属性）定义为 TMSF 格式时，可以通过 Win32 的 MCI_TMSF_FRAME、MCI_TMSF_MINUTE、MCI_TMSF_SECOND、MCI_TMSF_TRACK 四个宏把 Position 的值分解为轨道、分钟、秒、帧四个值。

但是，并非所有的媒体类型都支持 TMSF 格式。所以在本程序中使用了一种简单的方式来统一处理各种媒体的时间显示：即利用 TTimer 定时器并定义 mMinutes，mSeconds 变量，在程序中自己计算进度时间值。

响应 OnTimer 事件，Timer1Timer 函数内容如下：

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    if (MediaPlayer1->Mode==mpStopped){
        Timer1->Enabled=false;
    }
    DisplayTime(mMinutes/60,mMinutes,mSeconds);
    mSeconds++;
    mMinutes+=mSeconds/60;
    mSeconds%=60;
}
```

其中 DisplayTime 函数用于把时间显示于面板上，本程序采取较为简单的方法——直接从资源读出：

```
void TForm1::DisplayTime(int Hour, int Minutes, int Second)
{
    Image9->Picture->Bitmap
        ->LoadFromResourceName(0,"NUM"+IntToStr(Hour));
    Image11->Picture->Bitmap
        ->LoadFromResourceName(0,"NUM"+IntToStr(Minutes/10));
    Image12->Picture->Bitmap
        ->LoadFromResourceName(0,"NUM"+IntToStr(Minutes%10));
    Image14->Picture->Bitmap
        ->LoadFromResourceName(0,"NUM"+IntToStr(Second/10));
    Image15->Picture->Bitmap
        ->LoadFromResourceName(0,"NUM"+IntToStr(Second%10));
}
```

以上方法是一种有效的处理。但一般来说，更加自然的思路是定义一个 TBitmap 类型的数组如 Graphics::TBitmap *numBmps[10]，在初始化时将数字位图装入内存 TBitmap 变量，并在 DisplayTime 函数中用 TCanvas 的 Draw 方法显示之。当然，将数字位图装入 TImageList 组件，并使用 TImageList 组件的 Draw 方法绘制数字也是不错的选择。

7.5.3 实现无标题面板的拖动

为了实现控制面板的风格，我们将面板窗体的 BorderStyle 设为 bsNone，这样就屏蔽掉了窗体的标题栏，从而用户也就不能通过标题栏来拖动窗体。这显然带来了某种不方便。为了解决这个问题，我们需要编程实现拖动功能。

一种方法是“欺骗”系统，让它认为选中的是窗体标题行，这种方法通过响应 WM_NCHITTEST 消息来移动窗体。 Windows 系统发送 WM_NCHITTEST 消息来确定鼠标操作是否发生在窗体的客户区，或边框的特殊区上（非客户区）。如果 Windows 发现用户单击了窗体标题，系统将移动窗体，单击了窗体边框，则系统将开始改变窗体大小。编程中我们可以通过声明一个自定义事件，拦截 WM_NCHITTEST 消息。并且在响应函数中更改消息，使得系统认为选中的是标题行。这样做的结果是整个窗体都变成了“标题行”，从而用户单击任何一处都可以进行拖动。

在本程序中，我们采用一种易于实现并且效果更好的方法（这种方法被很多专业播放器所使用，如 Winamp），其主要原理是当无标题栏窗口进行鼠标拖动时，存在一个明显事实——鼠标在窗口中的坐标始终不变。 所以如果能够在鼠标移动过程中，通过改变窗口在桌面上的坐标，始终保持鼠标的相对坐标不变, 即可实现鼠标的拖动效果；在具体的程序设计中，先在 OnMouseDown 事件中记录鼠标位置，而在 OnMouseMove 事件中根据鼠标的移动距离，实时修改窗体 Form 的 Top 及 Left 值，即可实现窗口的鼠标拖动操作。

在 OnMouseDown 事件中记录鼠标位置，并标记鼠标拖动开始：

```
void __fastcall TForm1::Image1MouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
    isDraging=true;
    xx=X;
    yy=Y;
}
```

MouseMove 事件中修改窗体 Form 的 Top 及 Left 值，实现目标窗口的鼠标拖动操作。OnMouseMove 响应函数如下：

```
void __fastcall TForm1::Image1MouseMove(TObject *Sender, TShiftState Shift,int X, int Y)
{
    if (isDraging) {
        Form1->Left+=X-xx;
        Form1->Top+=Y-yy;
    }
```

```
}
```

最后在 MouseUp 事件中解除拖动状态：

```
void __fastcall TForm1::Image1MouseUp(TObject *Sender, TMouseButton Button,TShiftState Shift, int X,
int Y)
{
        isDraging=false;
}
```

7.5.4 实现音量调整功能

C++ Builder 的 TMediaPlayer 组件并不支持音量调整功能，然而音量调整又是一个多媒体播放程序所必须的。要实现音量调整，可以借助 Windows 的多媒体 API 函数。

音量调整窗体如图 7-9 所示。

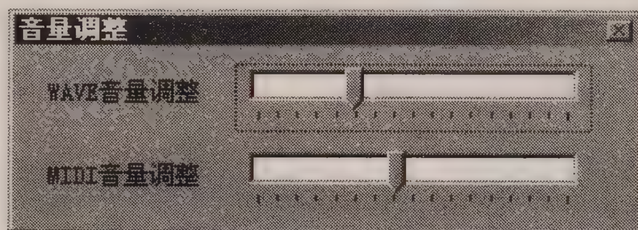


图 7-9 音量调整窗体

有 4 个多媒体 API 和音量调整直接相关：waveOutGetVolume、midiOutGetVolume、waveOutSetVolume 和 midiOutSetVolume，这些函数分别用于控制 Wave 音量和 MIDI 音量，函数的声明如下：

```
MMRESULT waveOutGetVolume(  // 获取指定波形输出设备的音量值
        HWAVEOUT hwo,
        LPDWORD pdwVolume
  );
MMRESULT midiOutGetVolume(  // 获取指定内部 MIDI 合成器设备音量值
        HMIDIOUT hmo,
        LPDWORD lpdwVolume
  );
MMRESULT waveOutSetVolume(  // 设置指定波形输出设备的音量值
        HWAVEOUT hwo,
        DWORD dwVolume
  );
MMRESULT midiOutSetVolume(  // 设置指定内部 MIDI 合成器设备音量值
        HMIDIOUT hmo,
        DWORD dwVolume
  );
```

在 BCBPlayer 程序中，我们通过两个 TTrackBar 组件显示和动态调整 Wave 设备和 MIDI 设备的音量。具体实现函数如下。

Source 实现波形输出和 MIDI 输出音量调整

```
void __fastcall TForm4::FormShow(TObject *Sender)
{
        DWORD Volume1,Volume2;
        waveOutGetVolume(0,&Volume1);
        TrackBar1->Position=Volume1;
        midiOutGetVolume(0,&Volume2);
        TrackBar2->Position=Volume2;
}
//---------------------------------------------------------------------------
void __fastcall TForm4::TrackBar1Change(TObject *Sender)
{
        waveOutSetVolume(0,TrackBar1->Position);
}
//---------------------------------------------------------------------------
void __fastcall TForm4::TrackBar2Change(TObject *Sender)
{
        midiOutSetVolume(0,TrackBar2->Position);
}
//---------------------------------------------------------------------------
```

7.6 MCI 与高级多媒体性能

在 7.4 节和 7.5 节中，我们利用 C++ Builder 提供的 TMediaPlayer 组件完成了一个完整的多媒体播放器。虽然 BCBPlayer 播放器已经相当出色，但事实上它与一些著名的多媒体播放器程序依然有相当距离。TMediaPlayer 组件固然功能强大，但在特定情况下，仍然需要直接依靠 MCI 来控制多媒体操作。

以上的论断是基于这样一种考虑，在 C++ Builder 中，由于多媒体的控制器非常简单，然而我们会发现它的能力受到某些限制。如果希望对较多的低级函数进行操作，就不得不应用 C++ Builder 去挖掘那些多媒体背后隐藏的深层次内容。从这一方面来说，TMediaPlayer 组件可能恰恰在程序员与所需获得信息之间起了阻碍的作用。而另一方面，如果应用程序对效率或资源占用方面有较高的要求，TMediaPlayer 组件也不是一个很好的选择。

7.6.1 TMediaPlayer 组件

如果放弃使用 TMediaPlayer，那还有什么其他选择呢？

Microsoft 提供了三种不同的界面使程序打开多媒体设备，其中两种是属于 MCI（Media Control Interface）部分的，第三种则是严格并有较高要求的低级 API。

对于 MCI 部分，提供两种不同的编程接口。这两种编程接口是命令串（Command-String）接口和命令消息（Command-Message）接口。命令串接口是为高级语言如 VB 提供支持而设计的基于字符串的接口，其主要函数是 mciSendString，在 C++ Builder 中可以调用这些函数，但命令消息（Command-Message）接口是更好的，我们更加倾向使用这种方法。

在这一节，我们利用 MCI 完成多媒体操作所实现的功能与利用 TMediaPlayer 组件是一致的，但这并不意味着 MCI 的能力仅仅如此。如果想编制有高级要求的多媒体程序，参考 Microsoft 提供的资料并通读 mmsystem.h 头文件是有必要的。

一般编程任务中，往往只需要播放某些音效。这种情况使用简洁有效的 sndPlaySound 函数更为合适。此函数的原型如下：

```
BOOL sndPlaySound( LPCSTR lpszSound , UINT fuSound );
```

其中字符串类型 lpszSound 参数为播放的 Wave 音频文件的文件名。fuSound 参数用以设置播放标志。有以下的定义值：

SND_ASYNC	异步播放设定。
S ND_LOOP	循环播放设定。
SND_NODEFAULT	如果 lpszSound 指定的音频文件媒没有找到，将不播放默认的声音。
SND_SYNC	同步播放设定。
SND_MEMORY	播放内存中的声音。此时第一个参数为声音资源的句柄的一个字符串类的强制转换。

例如可用 sndPlaySound(“sound.wav”,SND_LOOP|SND_SYNC)循环播放名为“sound.wav”的波形文件；而 sndPlaySound(wav_handle, SND_MEMORY|SND_SYNC)的调用方式实现播放一段内存中的 Wave 声音。

7.6.2 命令消息接口与 mciSendCommand 语言

MCI 的重要特点（优点）是它带给程序员一个一般的接口，而且并不会随着硬件变更而变化。应用 MCI 虽然较利用 C++ Builder 组件编程繁杂一些，但其基本方法仍是简单的。较之低级 API，MCI 接口不但使程序员不必研究某个特定设备如何工作的技术细节，而且可以用几乎一致的方式处理各种媒体文件。

MCI 的命令串（Command-String）接口在 7.1 节中有过简单的介绍。这里我们将关注更加灵活更有效率的命令消息（Command-Message）接口，命令消息接口很大程度上依赖于 mciSendCommand 函数。也许听起来觉得有些局限，但事实并非如此。深入接触命令消息接口编程后，你将了解这个系统所呈现的巨大的灵活性。

mciSendCommand 函数的原型如下：

```
MCIERROR mciSendCommand(
  MCIDEVICEID IDDevice,
```

```
    UINT uMsg,
    DWORD fdwCommand,
    DWORD dwParam
   );
```

下面具体介绍这四个参数的含义。IDDevice 参数用来识别问题中的设备的句柄或 ID 号。当我们未对媒体设备进行打开操作时，由于这个设备尚不具有 ID 号，则在这个参数的位置写 0。此后，这个设备的 ID 号会随着打开操作的完成传递过来。

uMsg 参数是整个接口的关键，作为一个函数的消息，指定一个特定的命令。下面是 15 个最常见的消息及其含义：

MCI_OPEN	打开设备。
MCI_CLOSE	关闭设备。
MCI_PLAY	用设备播放乐曲或乐曲的一部分。
MCI_SEEK	媒体前移或后移。
MCI_STOP	停止播放或录制。
MCI_INFO	查询正在工作的硬件类型。
MCI_GETDEVCAPS	查询设备能力。
MCI_SET	变更设备设置。
MCI_STEP	单步执行视频媒体的一帧或多帧。
MCI_RECORD	向一个设备录制。
MCI_SAVE	存储媒体到文件。
MCI_STATUS	查询设备是否在播放或暂停等。
MCI_LOAD	从文件读取，Digital-video 和 video-overlay 对此标记提供支持。
MCI_RESUME	重新播放或录制。

第三个参数 fdwCommand 是一系列的标志和记录，可以传递给一些调试工具。例如，MCI_PLAY 消息有四个这样的标志，可以把这 4 个符号用按位或形成第三个参数。

MCI_NOTIFY	完成时发送 MM_NOTIFY 消息。
MCI_WAIT	在返回前进行操作。
MCI_FROM	起始位置已确定。
MCI_TO	终点位置已确定。

假设此参数为 MCI_FROM|MCI_TO，则它们告知 MCI 起点和终点将在最后的参数中指定。如：

```
mciSendCommand(DeviceID,MCI_PLAY,MCI_FROM|MCI_TO,DWORD(&Info));
```

第四个参数 dwParam 是一个指向一个结构的指针。根据正在发送的消息，由 mciSendCommand 传送的结构作相应的变化。如在使用 MCI_PLAY 消息时，其结构是这样的：

```
typedef struct tagMCI_PLAY_PARMS {
   DWORD   dwCallback;
   DWORD   dwFrom;
   DWORD   dwTo;
} MCI_PLAY_PARMS;
```

MCI_PLAY_PATMS 结构与 MCI_PLAY 消息有着清晰紧密的联系。其他消息也都有它们自己独立的独一无二的结构对应。例如 MCI_OPEN 消息是与 MCI_OPEN_PARMS 对应的等等。关于 mciSendCommand 函数定义的消息和结构，可以在 mmsystem.h 头文件中查到。

mciSendCommand 函数有一个返回值，它作为错误的标记数字保存在一个长整型数的低位字里。通过 mciGetErrorString 函数可处理这个错误值：该函数提供 mciSendCommand 函数任意返回值的字符串类型的错误信息。

7.6.3 播放文件和录制声音

本节展示了应用 MCI 进行多媒体编程的两个应用程序：PlayWave 和 RecordWave。PlayWave 程序展示了利用 MCI 进行高性能 Wave 文件播放的能力。其界面如图 7-10 所示。

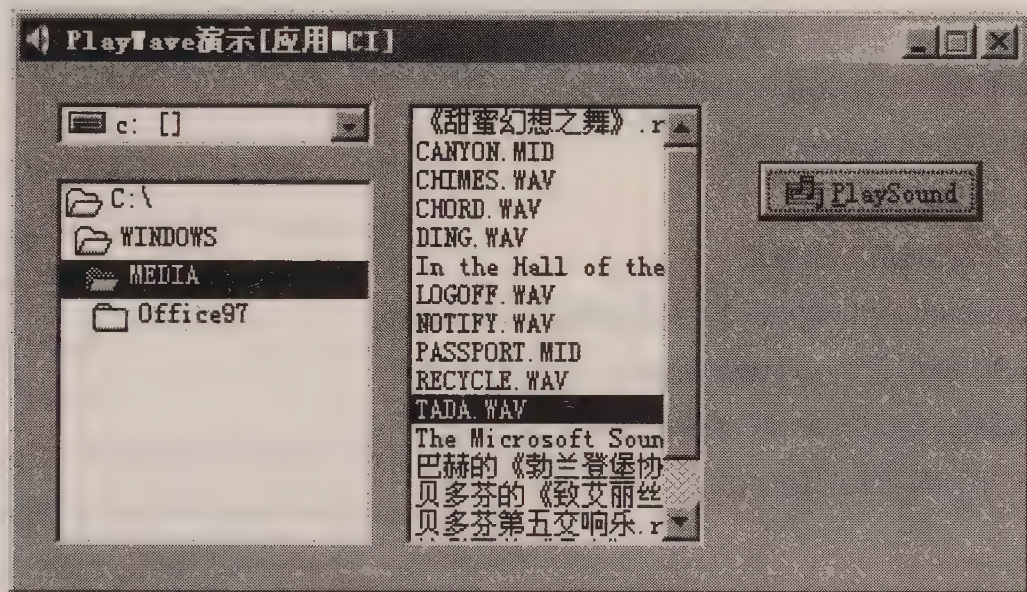


图 7-10　PlayWave 演示程序

在 PlayWave 和 RecordWave 演示程序中，包含几个通用的 MCI 媒体播放函数，可以不加修改的应用于任何程序之中，有一定的参考价值。其中 mciError 函数是一个通用的 MCI 错误处理函数，如下：

```
bool TForm1::mciError(long error, char *caller)
{
    char msg[100];
    if (error!=0){
        mciGetErrorString(error,msg,100);
        Application->MessageBox(msg,caller,ID_OK);
        return true;
    }
    return false;
}
```

PlayWave 函数是这个程序的核心函数，实现代码如下：

Source 使用 MCI 实现 PlayWave 函数

```
//---------------------------------------------------------------------------
void TForm1::PlayWave(char * filename)
{
        MCI_OPEN_PARMS mciOpen;
        MCI_PLAY_PARMS mciPlay;
        DWORD flags;
        mciOpen.lpstrDeviceType="waveaudio";
        mciOpen.lpstrElementName=filename;
        flags=MCI_OPEN_ELEMENT|MCI_OPEN_TYPE;
        mciError(mciSendCommand(0,MCI_OPEN,flags,
              DWORD(&mciOpen)),"OpenWave");
        DeviceID=mciOpen.wDeviceID;
        mciPlay.dwFrom=0;
        flags=MCI_FROM|MCI_WAIT;
        mciError(mciSendCommand(DeviceID,MCI_PLAY,flags,
              DWORD(&mciPlay)),"PlayWave");
        mciError(mciSendCommand(DeviceID,MCI_CLOSE,
              0,DWORD(NULL)),"Close");
}
//---------------------------------------------------------------------------
```

函数中包含了 Wave 文件的打开、播放、关闭，这三个操作都是通过 mciSendCommand 函数完成的。在函数中，mciSendCommand(0,MCI_OPEN,flags,DWORD(&mciOpen)句试图打开一个文件，并把 MCI_OPEN 传入该函数。在打开 WAV 设备之后，通过 mciOpen 获取对设备 ID 的控制：DeviceID=mciOpen.wDeviceID；假如一切成功的话，则播放文件。

```
mciPlay.dwFrom=0;
flags=MCI_FROM|MCI_WAIT;
mciError(mciSendCommand(DeviceID,MCI_PLAY,flags,DWORD(&mciPlay)),"Play");
```

flags 参数设定为 MCI_FROM|MCI_WAIT，MCI_FROM 标志声明程序将给出文件开始播放的地方，本例中由 mciPlay.dwFrom=0 置为零点。MCI_WAIT 标志声明了在播放结束前不希望该函数返回。

假定指定的 Wave 文件已成功的播放了，那么剩下的步骤是关闭设备：

```
mciError(mciSendCommand(DeviceID,MCI_CLOSE,0,DWORD(NULL)),"Close");
```

关闭设备时第四个参数并不接受特殊的返回，也就没有送入任何标志集合的必要。因此这个命令容易实现，而且关闭任何设备均可用同一模式。

另一个程序是与 PlayWave 类似的 RecordWave 程序，它能够通过麦克风将声音存入文件。其核心函数是名为 RecordWave 的通用函数。

RecordWave 演示程序如图 7-11 所示。

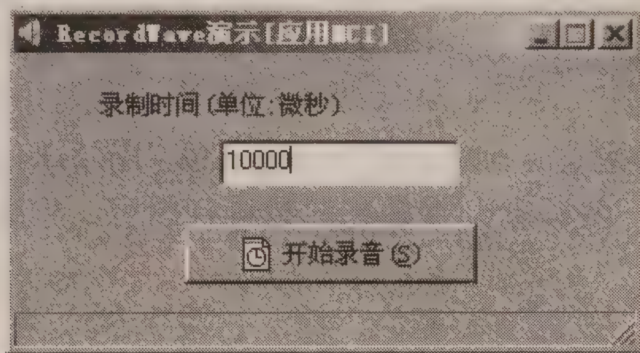


图 7-11 RecordWave 演示程序

Source 使用 MCI 实现 RecordWave 函数

```
//---------------------------------------------------------------------------
void TForm1::RecordWave(long time,char * filename)
{
    MCI_OPEN_PARMS mciOpen;
    MCI_RECORD_PARMS mciRecord;
    MCI_SAVE_PARMS mciSave;
    DWORD flags;
    long DeviceID;
    StatusBar1->Panels->Items[0]->Text="Recording...";
    mciOpen.lpstrDeviceType="waveaudio";
    mciOpen.lpstrElementName="";
    flags=MCI_OPEN_ELEMENT|MCI_OPEN_TYPE;
    mciError(mciSendCommand(0,MCI_OPEN,flags,DWORD(&mciOpen)),
        "OpenWave");
    DeviceID=mciOpen.wDeviceID;
    mciRecord.dwTo=time;
    flags=MCI_TO|MCI_WAIT;
    mciError(mciSendCommand(DeviceID,MCI_RECORD,flags,
        DWORD(&mciRecord)),"RecordWave");
    StatusBar1->Panels->Items[0]->Text="Finished...";
    mciSave.lpfilename=filename;
    flags=MCI_SAVE_FILE|MCI_WAIT;
    mciError(mciSendCommand(DeviceID,MCI_SAVE,flags,DWORD(&mciSave)),
        "SaveToFile");
    mciError(mciSendCommand(DeviceID,MCI_CLOSE,flags,DWORD(NULL)),
        "Close");
}
//---------------------------------------------------------------------------
```

RecordWave 程序较 PlayWave 稍复杂，但在此作者并不打算分析 RecordWave 程序的细节。相信在理解了 PlayWave 程序的运行机制之后，理解 RecordWave 程序是不成问题的。

通过以上两个例子可以看出，应用 MCI 处理多媒体是强大并且灵活的，它代表了较深层次的多媒体编程技术。然而实际上，mciSendCommand 接口背后的基本理论并不难以理解。虽然我们不得不花一段时间对一些标志、常量和参数进行研究，但包含的基本概念和基本方法是简单的。

7.7 底层多媒体 API

除了 MCI 接口之外，Windows 还提供一整套底层多媒体 API 函数为多媒体编程提供更多细节控制。一般来说在使用底层多媒体 API 的时候，我们想做的已经不是如何简单的将声音或视频文件播放出来了，而是要进行深入的控制或处理。例如，想编写一个类似 Windows 录音机的程序，它在播放或录制 Wave 时可以将声音波形以图像显示出来，又如想将两个 Wave 音频同时播放（混音），这些高级功能都必须借助 Windows 的底层多媒体 API。

7.7.1 RIFF 文件

使用底层多媒体 API 往往需要编程人员对多媒体文件、多媒体播放机制有一定了解。Mircosoft 定义了 RIFF 格式（Resource Interchange File Format，资源交换文件格式）作为多媒体文件格式。在 Windows 中，包括波形音频文件（Wave），AVI 动画剪辑，调色板文件和 RMID 格式的 MIDI 音频文件都采用了 RIFF 格式。RIFF 格式是带标签的文件结构，即文件由一系列不规则的部分组成，每部分通过标签隔开。

RIFF 文件的基本构造元素称为块（Chunk），每个标签开始一个块，每个块包含以下 3 个部分：

- 标签。由 4 个字符组成的代码，如“RIFF”、“DATA”等，指定块的标识符。
- 尺寸域。指定块的数据域大小的双字。
- 数据域。存储该块的实际数据，可能包含若干子块。

Wave 文件格式可由图 7-12 表示。

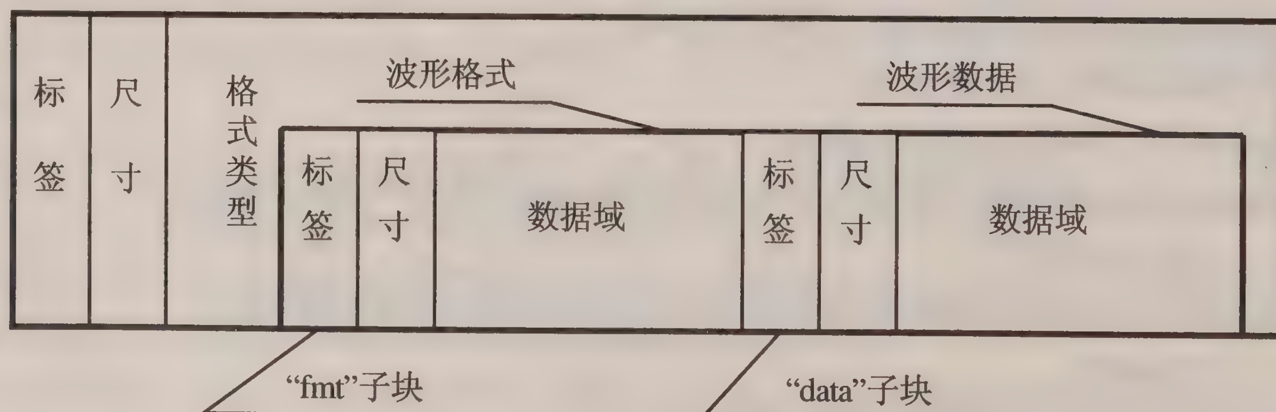


图 7-12 Wave 文件格式

Wave 文件是一种非常简单的 RIFF 文件，其结构只有两层。其中，fmt 子块和 data 子块存储波形音频的实际数据——fmt 子块用于存储波形格式信息，包括声道数、采样率、样本位数等；data 子块用于存储波形数据信息。

读写各种 RIFF 文件需要使用 RIFF 文件 I/O 函数，其中重要的函数包括：mmioOpen、mmioRead、mmioSeek、mmioWrite、mmioFOURCC、mmioAscend、mmioDescend、mmioClose 等，这些函数的功能简要描述如下：

mmioOpen	打开文件进行读写操作，返回打开文件句柄。
mmioRead	从一打开文件读取指定字节数的数据。
mmioSeek	在打开的文件中改变当前文件指针。
mmioWrite	向一打开文件写入指定字节数的数据。
mmioFOURCC	将四个字符转换为 4 字符码。
mmioAscend	升出指定块，表示将文件指针移到数据末尾处。
mmioDescend	降入指定块，表示将文件指针移到该块开始处。
mmioClose	关闭利用 mmioOpen 打开的文件。

7.7.2 使用低级 API 实现 Wave 播放

使用低级 API 实现 Wave 播放涉及的函数较多，可以分为两步实现——波形音频文件的打开和通过波形输出设备播放。即，首先打开一个.WAV 文件并将文件中的数据加载到内存中去，它完成下列步骤：

(1) 打开波形 RIFF 文件。

(2) 寻找 WAVE 块，并判断该文件是否为有效 Wave 文件。

(3) 定位 fmt 子块，确认声音以一种合适的形式存在。

(4) 定位 data 子块，并将该块数据加载到内存缓冲区。

(5) 关闭波形文件。

其次调用相关函数实现内存中数据的播放。为了正确播放 Wave，首先调用了 waveOutOpen 以打开波形输出设备，而后调用 waveOutPrepareHeader 函数准备音频数据块。最后调用 waveOutWrite 函数将内存缓冲区的音频数据实际输出。在音频播放结束后，必须调用 waveOutClose 和 waveOutUnprepareHeader 函数关闭输出设备。

这里我们通过一个实例程序演示如何使用音频底层 API。该实例可以打开一个 Wave 文件，并支持正常播放和逆序播放。可以想象，如果调用高级 API，逆序播放这样的功能是绝无可能实现的。而利用底层 API，实现逆序播放轻而易举（见图 7-13）。

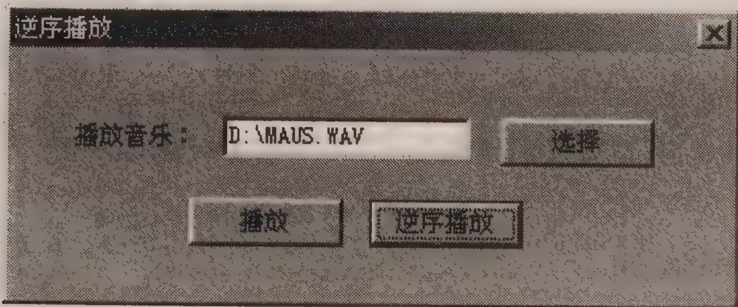


图 7-13 底层多媒体 API 实现 Wave 的逆序播放

具体程序如下：

Source 底层多媒体 API 实现 Wave 播放

```
void __fastcall TForm1::ReversePlay(char *FileName,bool bReverse)
{
    HMMIO hmmio;
    MMCKINFO mmckinfoParent;
    MMCKINFO mmckinfoSubchunk;
    DWORD dwFmtSize;
    DWORD dwResult;
    HANDLE hFormat;
    WAVEFORMATEX *pFormat;
    DWORD dwDataSize;
    HPSTR hpch1, hpch2;
    WORD wBlockSize;
    HANDLE hData;
    // 调用多媒体文件 I/O 函数打开 RIFF 文件
    if(!(hmmio = mmioOpen(FileName, NULL, MMIO_READ | MMIO_ALLOCBUF)))
        return;
    // 定位"WAVE"块
    mmckinfoParent.fccType = mmioFOURCC('W', 'A', 'V', 'E');
    if (mmioDescend(hmmio, &mmckinfoParent, NULL, MMIO_FINDRIFF)){
        mmioClose(hmmio, 0);
        return;
    }
    // 定位"fmt"子块
    mmckinfoSubchunk.ckid = mmioFOURCC('f', 'm', 't', ' ');
    if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent,
        MMIO_FINDCHUNK)){
        mmioClose(hmmio, 0);
        return;
    }
    // 得到格式块大小，分配相应内存
    dwFmtSize = mmckinfoSubchunk.cksize;
    hFormat = LocalAlloc(LMEM_MOVEABLE, LOWORD(dwFmtSize));
    if (!hFormat){
        mmioClose(hmmio, 0);
        return;
    }
```

```
pFormat = (WAVEFORMATEX *) LocalLock(hFormat);
if (!pFormat){
    LocalFree( hFormat );
    mmioClose(hmmio, 0);
    return;
}
// 读取"fmt"块
if (mmioRead(hmmio, (HPSTR) pFormat, dwFmtSize) != (LONG) dwFmtSize){
    LocalUnlock( hFormat );
    LocalFree( hFormat );
    mmioClose(hmmio, 0);
    return;
}
// 确定该文件确实为 WAVE 文件
if (pFormat->wFormatTag != WAVE_FORMAT_PCM){
    LocalUnlock( hFormat );
    LocalFree( hFormat );
    mmioClose(hmmio, 0);
    return;
}
if (waveOutOpen(&hWaveOut, WAVE_MAPPER, pFormat, 0, 0L,WAVE_FORMAT_QUERY)){
    LocalUnlock( hFormat );
    LocalFree( hFormat );
    mmioClose(hmmio, 0);
    return;
}
// 升出"fmt"块
mmioAscend(hmmio, &mmckinfoSubchunk, 0);
// 寻找数据子块，确定数据块大小
mmckinfoSubchunk.ckid = mmioFOURCC('d', 'a', 't', 'a');
if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent,MMIO_FINDCHUNK)){
    LocalUnlock( hFormat );
    LocalFree( hFormat );
    mmioClose(hmmio, 0);
    return;
}
dwDataSize = mmckinfoSubchunk.cksize;
if (dwDataSize == 0L)
{
    LocalUnlock( hFormat );
```

```
        LocalFree( hFormat );
        mmioClose(hmmio, 0);
        return;
}
// 确认系统存在波形输出设备
if (waveOutOpen(&hWaveOut, WAVE_MAPPER,
        pFormat,(DWORD)Application->Handle,0, CALLBACK_WINDOW)){
        LocalUnlock( hFormat );
        LocalFree( hFormat );
        mmioClose(hmmio, 0);
        return;
}
wBlockSize = pFormat->nBlockAlign;
LocalUnlock( hFormat );
LocalFree( hFormat );
// 使用 GlobalAlloc 函数分配内存
lpData =(char *)GlobalAllocPtr(GMEM_MOVEABLE|GMEM_SHARE,dwDataSize);
if (!lpData){
        mmioClose(hmmio, 0);
        return;
}
// 读取波形数据子块
if(mmioRead(hmmio, lpData, dwDataSize) != (LONG) dwDataSize){
        GlobalFreePtr( lpData );
        mmioClose(hmmio, 0);
        return;
}
mmioClose(hmmio, 0);
// 反转波形数据
if (bReverse){
        hpch1 = lpData;
        hpch2 = lpData + dwDataSize - 1;
        while (hpch1 < hpch2){
                Exchange( hpch1,hpch2,wBlockSize );
                hpch1 += wBlockSize;
                hpch2 -= wBlockSize;
        }
}
// 建立数据块头结构
lpWaveHdr = (LPWAVEHDR)GlobalAllocPtr(GMEM_MOVEABLE | GMEM_SHARE,
```

```
                    (DWORD) sizeof(WAVEHDR));
        if (!lpWaveHdr){
            GlobalFreePtr( lpData );
            return;
        }
        // 准备音频数据块
        lpWaveHdr->lpData = lpData;
        lpWaveHdr->dwBufferLength = dwDataSize;
        lpWaveHdr->dwFlags = 0L;
        lpWaveHdr->dwLoops = 0L;
        if(waveOutPrepareHeader(hWaveOut, lpWaveHdr, sizeof(WAVEHDR))){
            CleanGlobal();
            return;
        }
        // 将数据块发送到音波形输出设备
        dwResult = waveOutWrite(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));
        if (dwResult != 0){
            waveOutUnprepareHeader( hWaveOut, lpWaveHdr, sizeof(WAVEHDR));
            CleanGlobal();
            return;
        }
}
HGLOBAL hWaveHeader=NULL;
LPWAVEHDR lpWaveHeader=NULL;
//---------------------------------------------------------------------------
bool OpenWaveFile(HWND hWnd,AnsiString FileName)
{
        MMCKINFO mmChildInfo,mmParentInfo;
        DWORD dwFmtSize;
        // 调用多媒体文件 I/O 函数打开 RIFF 文件
        HMMIO hmmio = mmioOpen(FileName.c_str(),NULL,MMIO_ALLOCBUF|MMIO_READ);
        if (hmmio==NULL){
            MessageBox(hWnd,"打开波形文件失败!","错误",MB_ICONEXCLAMATION|MB_OK);
            return false;
        }
        // 定位"WAVE"块
        mmParentInfo.fccType=mmioFOURCC('W','A','V','E');
        if (mmioDescend(hmmio,&mmParentInfo,NULL,MMIO_FINDRIFF))
            goto Error;
        // 定位"fmt"子块
```

```
        mmChildInfo.ckid=mmioFOURCC('f','m','t',' ');
        if (mmioDescend (hmmio,&mmChildInfo,&mmParentInfo,MMIO_FINDCHUNK))
            goto Error;
        // 得到格式块大小，分配相应内存
        dwFmtSize = mmChildInfo.cksize;
        hFormat=::LocalAlloc(GMEM_MOVEABLE,dwFmtSize);
        pFormat=(PCMWAVEFORMAT*)::LocalLock(hFormat);
        // 读取"fmt"块
        if (mmioRead(hmmio,(LPSTR)pFormat,dwFmtSize)!=(LRESULT)dwFmtSize){
            LocalUnlock(hFormat);
            LocalFree(hFormat);
            goto Error;
        }
        // 升出"fmt"块
        mmioAscend(hmmio,&mmChildInfo,0);
        // 寻找数据子块
        mmChildInfo.ckid=mmioFOURCC('d','a','t','a');
        if (mmioDescend(hmmio,&mmChildInfo,&mmParentInfo,MMIO_FINDCHUNK)){
            LocalUnlock(hFormat);
            LocalFree(hFormat);
            goto Error;
        }
        dwDataSize=mmChildInfo.cksize;
        return true;
    Error:
        MessageBox(hWnd,"打开波形文件失败!","错误",MB_ICONEXCLAMATIONIMB_OK);
        mmioClose(hmmio,0);
        return false;
    }
    //-----------------------------------------------------------------------
    void PlayWaveFile(HWND hWnd)
    {
        HWAVEOUT hWaveOut;
        UINT wResult;
        // 确认系统存在波形输出设备
        wResult=waveOutOpen(NULL, WAVE_MAPPER,(LPWAVEFORMATEX)pFormat,
            0L,0L,WAVE_FORMAT_QUERY);
        if (wResult) {
            LocalUnlock(hFormat);
            LocalFree(hFormat);
```

```
        goto Error1;
}
// 使用 GlobalAlloc 函数分配内存
hData=::GlobalAlloc(GMEM_MOVEABLE|GMEM_SHARE,dwDataSize);
lpData=(HPSTR)GlobalLock(hData);
// 读取波形数据子块
if (mmioRead(hmmio,(HPSTR)lpData,dwDataSize)!=(LRESULT)dwDataSize){
        GlobalUnlock(hData);
        GlobalFree(hData);
        goto Error1;
}
// 打开波形设备
wResult =waveOutOpen((LPHWAVEOUT)&hWaveOut,WAVE_MAPPER,
        (LPWAVEFORMATEX)pFormat,(DWORD)hWnd,NULL,CALLBACK_WINDOW);
if (wResult){
        LocalUnlock(hFormat);
        LocalFree(hFormat);
        goto Error1;
}
LocalUnlock(hFormat);
LocalFree(hFormat);
// 建立数据块头结构
hWaveHeader=GlobalAlloc(GMEM_MOVEABLE|GMEM_SHARE,sizeof(WAVEHDR));
lpWaveHeader=(LPWAVEHDR)GlobalLock(hWaveHeader);
lpWaveHeader->lpData=lpData;
lpWaveHeader->dwBufferLength=dwDataSize;
lpWaveHeader->dwFlags=0L;
lpWaveHeader->dwLoops=0L;
// 准备音频数据块
wResult=waveOutPrepareHeader(hWaveOut,lpWaveHeader,sizeof(WAVEHDR));
if (wResult!=0){
        GlobalUnlock(hData);
        GlobalFree(hData);
        GlobalUnlock(hWaveHeader);
        GlobalFree(hWaveHeader);
        goto Error1;
}
// 将数据块发送到音频波形输出设备
wResult=waveOutWrite((HWAVEOUT)hWaveOut,lpWaveHeader,sizeof(WAVEHDR));
if (wResult!=0){
```

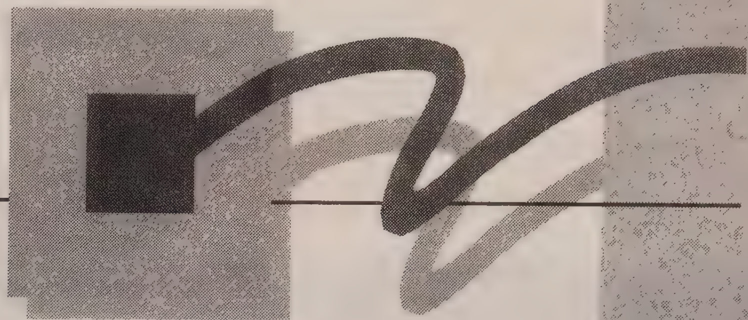
```
            waveOutUnprepareHeader((HWAVEOUT)hWaveOut,
                  lpWaveHeader,sizeof(WAVEHDR));
            MessageBox(hWnd,"波形文件数据块输出失败!","错误",
                  MB_ICONEXCLAMATION|MB_OK);
            GlobalUnlock(hData);
            GlobalFree(hData);
            GlobalUnlock(hWaveHeader);
            GlobalFree(hWaveHeader);
            mmioClose(hmmio,0);
            waveOutClose(hWaveOut);
            return;
      }
      return;
Error1:
      MessageBox(hWnd,"波形文件数据块初始化失败!","错误",
            MB_ICONEXCLAMATION|MB_OK);
      mmioClose(hmmio,0);
      return;
}
//---------------------------------------------------------------------------
```

第8章

俄罗斯方块游戏

——VCL 游戏编程与实用技术

程序规划和算法设计
逐步实现程序的具体模块
创建帮助系统
INI 文件支持和 Splash 屏幕
编程实现游戏手柄支持



本章导读

本章内容是 VCL 编程技术的综合应用，我们将开发一个真正完整的 Windows 游戏程序——俄罗斯方块游戏。该游戏程序支持游戏手柄操作，并且具备 Windows 游戏所需的所有要素——背景音乐及音效、英雄榜系统、帮助系统、可更换的背景图案、良好的操作方式（键盘和游戏杆）。这一切都由 C++ Builder 的 VCL 所实现。

本章内容包括:

- 对于游戏实例的规划和实现、程序结构的分析，并对隐藏在创建和设计 Windows 程序之后的某些原则作一些讨论。
- 逐步实现程序的具体模块。
- 讲述 Windows 编程的实用技术。包括帮助系统的建立，溅出屏幕（Splash Screen）和 INI 初始化文件的使用。
- 高级话题。为“俄罗斯方块”游戏提供游戏手柄支持。

游戏编程领域充满了新奇与乐趣。应用 C++ Builder 的强大功能，当你出色地完成了所设想的游戏程序时，那种美好的感觉是难以言喻的。

8.1 实例创建分析

在本章中，读者将看到"俄罗斯方块"程序最终版本的实现过程。这个程序的雏形是笔者用 JavaScript+HTML 编写的一个俄罗斯方块游戏。现在用 C++ Builder 重写了它，并将这个程序作为 VCL 综合应用的实例程序，该程序界面如图 8-1 所示。

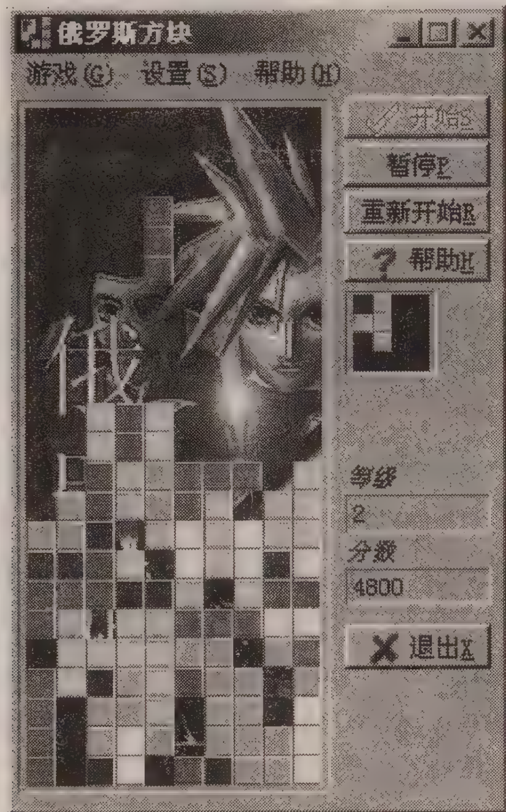


图 8-1 俄罗斯方块游戏程序

选择一个游戏程序作为应用 VCL 技术编程的综合实例是基于以下考虑：

- 游戏程序具有较为复杂的逻辑结构。"俄罗斯方块"程序不仅阐明了它自己的任务，同时表明了一个较大规模的程序的策划、求精和实现问题。
- 游戏程序是典型的实际应用程序。它可以暴露出现实世界中 Windows 程序员常会面临到的许多设计问题。
- 游戏实例可以增加读者对所学技术的兴趣。

对于现实游戏实例来说，它是一个较大规模的问题，因而整个程序的规划就显得尤为重要，一个良好的、结构合理的实现方法能够大大简化程序的编制，缩减程序代码，并为程序提供优良的性能和风格。在本章中，将就"俄罗斯方块"程序的规划和具体实现结构进行充分的分析。希望读者能从中学到对于较大规模问题如何下手，并条理清晰的、出色的完成它。

俄罗斯方块（Tetris）的游戏规则相信很多人都清楚，它是一个非常著名的游戏，这个游戏需要运用逻辑知识和敏锐的头脑去合理的放置不同类型的方块。

这里将要实现的游戏实例是俄罗斯方块游戏扩展版本，它将传统的 7 种类型方块扩展为 21 种。新的不规则的方块增加了游戏的难度和趣味性。

具体实现“俄罗斯方块”游戏程序包含以下步骤。

1. 创建应用程序窗体

因为涉及到程序直接绘制方块的问题，本程序中对主窗体中组件的位置和尺寸精度都有较高要求。解决的办法是在程序中定制主窗体关键组件的位置、尺寸和其他属性。所以在设计主窗体时只需将组件放在大致位置即可。

主窗体（TTetrisForm）的主要组件包括：

Image	TImage 类型组件。用于绘制游戏容器即游戏工作部分。
BGImage	TImage 类型组件。用于处理和存储背景图案，不可见。
NextImage	TImage 类型组件。绘制预览方块显示部分。
Timer1	TTimer 类型组件。驱动游戏进行的定时器组件。
Bevel1~ Bevel4	TBevel 类型组件。分别为游戏容器部分，预览显示部分，分数显示，等级显示提供一个三维风格变框。
Label3/Label4	TLabel 类型组件。分别用于显示等级和显示分数。
MediaPlayer1	TMediaPlayer 类型组件。用于播放背景音乐。
OpenPictureDialog1	TOpenPictureDialog 类型组件。用于支持背景图像的选择。

TTetries 工程中主要包括以下文件：

- 主窗体单元。类名：TTetrisForm，文件：Unit1.cpp，Unit1.h。
- 其他三个窗体：TAboutBox、TSplashForm 和 TScoreForm，分别用于显示“关于”信息、显示溅出屏幕和显示高分。其中 TSplashForm 设为 Avialable form。
- 帮助系统。文件：Tetri.hlp。

2. 数据组织

着手进行具体的编程工作，首先需要处理程序中的常量和变量，常量声明应包括方块形状的信息。在本范例程序中，没有采用一般的用 0-1 数组表示不同类型方块的做法，而是根据问题特点采用新的数据结构，这是本程序最大的特色所在，合理的数据组织大大简化了问题并带来高效率。

声明变量用于描述游戏中的各种状态。例如，游戏是否正在进行、音乐或音效是否打开等；同时声明用于描述游戏信息的变量，包括分数、级别等。

3. 定制窗体

在程序的初始化代码中，定制主窗体关键组件的位置和尺寸，以满足绘制需要。

4. 完成游戏部分代码

接下来具体实现游戏部分代码，首先应完成必要的绘制和擦除函数，包括在游戏容器绘制和预览框中绘制两个版本（因为游戏容器和预览框中方块的大小不一致）。其次，应完成方块移动和检测的函数。最后，在键盘事件响应函数和定时器事件响应函数中实现游戏过程。

5. 辅助功能

在程序的核心代码完成后，实现必要的辅助功能。包括背景音乐，音效的支持，更换背景图像的支持，以及高分纪录功能。同时为保存这些设置信息，加入对 INI 文件的支持。最后，为程序加入 About 窗体和一个溅出屏幕（Splash Screen）。

6. 创建帮助系统

在程序完成后，还将为“俄罗斯方块”游戏制作帮助系统，并在程序中实现帮助文件的支持。

8.2　实现俄罗斯方块程序的核心部分

基于各种编程语言的“俄罗斯方块”游戏例程很多，但那些程序往往采用了一种直接的数据结构，即采用 0-1 数组描述方块形状，并通过计算来旋转方块。但这样做造成了实现起来的复杂和艰难，以及程序代码的冗长。本程序则采用了不同的数据结构，这样在程序编制难度、代码复杂度方面都大大降低，同时获得较高的执行效率。

8.2.1　程序策划

考虑到每一种类型的方块都是由四个（或四个以下）小块组成的。我们只需纪录这四个小块在 4*4 范围内的坐标。例如图 8-2 中方块的信息可以记录为：21222333（每两个数字代表一个小块的横纵坐标）。



图 8-2　方块数据组织

这样，在绘制每一个方块时，只需分别将四个小块绘制出来即可。同时方块的移动，碰撞判断也都大大简化了。

基于上述的数据组织方式，概括阐述“俄罗斯方块”程序的核心部分实现策略如下：

- 所有 21 种方块的信息存放在 Style[21][40]的数组中，每种方块存储了四个方向的信息，这样就省去了方块旋转的处理。
- 构建绘制小块和擦除小块的函数，实现在游戏容器中 X、Y 位置处绘制指定颜色的小块。均有两个版本，分别用于游戏容器和预览框。
- GlassMap[GlassWidth][GlassHeight]数组纪录当前游戏容器内的非活动方块的状态。0 值表示此处无方块，1 ~ 7 分别对应了 7 种颜色，表示此处有方块以及方块的颜色。
- 用 ObjectX，ObjectY 变量描述当前方块所在的 4*4 区域的作上角方格的坐标，Patn 描述当前方块的类型。Rotat 变量描述当前方块的旋转状态，Patn 和 Rotat 一起用于在 Style 数组中找到具体形状信息。ObjectColor 表示当前操作方块的颜色。

- NextPatn、NextRotat 和 NextColor 分别表示下一个方块类型，方块旋转状态和方块颜色。
- 当当前活动方块（正在掉下来的方块）发生动作时，WriteObject 函数将方块正确地重画。
- 构建方块动作函数。包括 MoveDown、MoveLeft、MoveRight 和 Rotat。在函数中判断能否执行该动作（四个小块均在容器之内，并且没有与已有小块碰撞），如果是，则更新 ObjectX、ObjectY 及 Rotat，并调用 WriteObject 显示出结果。
- 如果在 MoveDown 函数中动作不能执行，则表示已经沉底，这时首先更新 GlassMap，再调用 ScanLines 函数处理消行，用 NextPatn、NextRotat、NextColor 更新 Patn、Rotat、ObjectColor，并判断此方块是否能在游戏容器最上方中间放置而不和容器内方块碰撞，若否，则游戏结束；若是，则调用 WriteNext 生成并画出下一个方块。
- 整个游戏由键盘事件 OnKeyDown 和定时器事件 OnTimer 驱动。上下左右键按下分别造成 Rotate、MoveDown、MoveLeft 和 MoveRight 动作，OnTimer 事件造成 MoveDown 动作。
- Start 函数执行初始化工作并开始游戏。其中首先调用 WriteNext 生成一个方块，并将当前方块数据更新为此方块；然后调用 WriteObject 将方块放入容器；最后再次调用 WriteNext 生成下一个方块。

以上就是此程序的大致逻辑过程。当然有关分数处理、升级处理等细节问题没有包括在内，因为这些部分是容易实现的，并且与程序总体的逻辑结构无太大关系。

8.2.2 数据处理和定制窗体

按以上分析具体实现各个模块就可以使游戏部分真正运作起来，但前提是完成必要的初始化工作。

1. 数据处理

在主窗体的头文件部分定义了一些重要的常量。

Source 俄罗斯方块游戏的常量定义

```
const int NormalWidth= 14, //游戏容器中方块的尺寸
      NormalHeight=14,
      SmallWidth= 9,       //预览框中方块的尺寸
      SmallHeight=9,
      Tp=  6,              //游戏容器左上角定位点
      Lft=  5,
      FieldWidth=4,        //游戏容器边界到实际工作区域边界的距离
      GlassWidth=10,// 游戏容器大小（容纳小块数）
```

```
        GlassHeight=23;
const TColor Colors[7]={clSilver,clRed,clLime,clBlue,
                clFuchsia,clAqua,clYellow};// 可用颜色
const AnsiString IniFileName="Tetris.ini";          // INI 文件名
// 方块数据。包括四个方向的信息，以--间隔
const char *Tetris[21]={"22233233--22233233--22233233--22233233--",
                "12212223--12212232--21222332--12222332--",
                "13222332--21223233--13222332--21223233--",
                "12222333--22233132--12222333--22233132--",
                "12223233--23313233--22233343--22232432--",
                "13233233--21222333--22233242--22323334--",
                "13233343--21222324--13233343--21222324--",
                "22223232--32323333--33332323--23232222--",
                "22222332--32322233--33333223--23232233--",
                "12123232--21212323--12123232--21212323--",
                "12222232--21222223--12222232--21222223--",
                "22223333--23233232--22223333--23233232--",
                "11132123--11123132--11132123--11123132--",
                "22222222--22222222--22222222--22222222--",
                "22223232--32323333--33332323--23232222--",
                "11222233--13222231--11222233--13222231--",
                "21212424--12124242--21212424--12124242--",
                "12222232--21222223--12222232--21222223--",
                "11132232--13212233--12223133--11222331--",
                "21223334--13233242--23243132--12223343--",
                "12223333--23243232--22223343--23233132--"};
```

同时，声明表示游戏状态的变量。

Source 俄罗斯方块游戏的主要变量定义

```
public:             // User declarations
    int Level;          // 当前级别
    long Score;         // 当前分值
    int ObjectX;        // 当前操作块的横向位置
    int ObjectY; // 当前操作块的纵向位置
    int NextPatn,NextRotat;      // 下一个方块的类型和旋转状态
    int Patn,Rotat;              // 当前方块的类型和旋转状态
    int NextColor,ObjectColor;   // 颜色，通过 Colors 常量数组取到实际颜色
    int OldX[4],OldY[4];         // 上一次动作时，四个小块的横纵位置
    char Style[21][40];          // 实际方块类型信息
```

```
int GlassMap[GlassWidth][GlassHeight];        // 当前游戏容器状态
int OldGlassMap[GlassWidth][GlassHeight];     // 消行时使用
bool MoveFlag;                     // 是否存在上次动作
bool isPlaying;                    // 游戏是否开始
bool PlaySnd,PlayMusic;            // 音乐，音效开关
AnsiString ImageFileName;   // 背景图像文件名
long HighScore[3];                 // 高分纪录
AnsiString HighName[3];            // 高分姓名纪录
TIniFile *IniFile;              // INI 文件变量
```

方块数组 Tetris 采用字符串数组的形式是为方便数据的创建、输入和修改。为方便程序操作，将数据转换到 Style 数组中。

```
void TTetrisForm::ReadStyle()
{
    for (int i=0; i<21; i++)
        for (int j=0; j<40; j++)
            if (Tetris[i][j]=='-') // '-'为无用信息
                Style[i][j]=0;
        else Style[i][j]=Tetris[i][j]-'0';  // 转换为整型
}
```

2. 定制窗体

在 OnCreate 响应函数中精确定制窗体组件的位置和尺寸。加入以下代码：

Source 运行时窗体组件位置和尺寸的定制

```
// 定制游戏容器的边框
Bevel3->Top= Tp-FieldWidth;
Bevel3->Left= Lft-FieldWidth;
Bevel3->Width= GlassWidth*NormalWidth+FieldWidth*2;
Bevel3->Height= GlassHeight*NormalHeight+FieldWidth*2;
// 定制游戏容器图像和背景图像
Image->Top=Tp;
Image->Left=Lft;
Image->Width=GlassWidth*NormalWidth;
Image->Height=GlassHeight*NormalHeight;
BGImage->Width=Image->Width;
BGImage->Height=Image->Height;
// 定制窗体和按钮
ClientWidth= Bevel3->Width+FieldWidth*3+Button1->Width;
ClientHeight= Bevel3->Height+FieldWidth*2;
```

```
Button1->Left = Bevel3->Width+FieldWidth*2;
Button2->Left = Button1->Left;
Button3->Left = Button1->Left;
Button5->Left = Button1->Left;
Button4->Left = Button1->Left;
// 定制得分框和等级框
Label1->Left= Bevel3->Width+FieldWidth*2;
Label2->Left= Label1->Left;
Bevel1->Left= Label1->Left;
Bevel1->Width= Button1->Width;
Bevel2->Left= Label1->Left;
Bevel2->Width= Button1->Width;
Label3->Left = Bevel1->Left+FieldWidth;
Label4->Left = Bevel1->Left+FieldWidth;
// 预览框和预览框内的图像
Bevel4->Top = Button5->Top+Button5->Height+4;
Bevel4->Left = Button5->Left;
Bevel4->Height = SmallHeight*4+4;
Bevel4->Width =SmallWidth*4+8;
NextImage->Top = Button5->Top+Button5->Height+6;
NextImage->Left = Button5->Left+4;
NextImage->Height = SmallHeight*4;
NextImage->Width =SmallWidth*4;
```

8.3 工作模块具体实现

8.3.1 核心工作模块

1. 绘制和清除

以下是游戏容器内绘制小块（单位方块）的函数 DrawBlock 和在预览框内绘制小块的函数 DrawSmallBlock。

Source 单位方块的绘制

```
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::DrawBlock(int X, int Y, TColor mColor)
{
```

```
        int X1,Y1,X2,Y2;
        X1 = X*NormalWidth;
        X2 = X1+NormalWidth;
        Y1 = Y*NormalHeight;
        Y2 = Y1+NormalHeight;
        Image->Canvas->Brush->Color=mColor;
        Image->Canvas->FillRect(Rect(X1+1,Y1+1,X2-1,Y2-1));
        Image->Canvas->Pen->Color = clGray;
        Image->Canvas->MoveTo(X1,Y1);
        Image->Canvas->LineTo(X1,Y2-1);
        Image->Canvas->LineTo(X2-1,Y2-1);
        Image->Canvas->Pen->Color = clWhite;
        Image->Canvas->LineTo(X2-1,Y1);
        Image->Canvas->LineTo(X1,Y1);
}
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::DrawSmallBlock(int X, int Y, TColor mColor)
{
        int X1,X2,Y1,Y2;
        X1 = X*SmallWidth;
        X2 = X1+SmallWidth;
        Y1 = Y*SmallHeight;
        Y2 = Y1+SmallHeight;
        NextImage->Canvas->Brush->Color = mColor;
        NextImage->Canvas->FillRect(Rect(X1+1,Y1+1,X2-1,Y2-1));
        NextImage->Canvas->Pen->Color = clGray;
        NextImage->Canvas->MoveTo(X1,Y1);
        NextImage->Canvas->LineTo(X1,Y2-1);
        NextImage->Canvas->LineTo(X2-1,Y2-1);
        NextImage->Canvas->Pen->Color = clWhite;
        NextImage->Canvas->LineTo(X2-1,Y1);
        NextImage->Canvas->LineTo(X1,Y1);
}
//---------------------------------------------------------------------------
```

由于游戏容器和预览框的尺寸已经定制好，所以可以容易地在合适位置绘制小块。在绘制小块时进行了颜色填充，并使用 TCanvas 类的 LineTo 函数绘制边框，使每一个小块看上去具有 3D 效果。

以下是擦除游戏容器内小块的函数 ClearBlock 和擦除预览显示框内小块的函数 ClearSmallBlock。

Source 单位方块的清除

```
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::ClearBlock(int X, int Y)
{
        int X1,Y1,X2,Y2;
        X1 = X*NormalWidth;
        X2 = X1+NormalWidth;
        Y1 = Y*NormalHeight;
        Y2 = Y1+NormalHeight;
        Image->Canvas->CopyRect(Rect(X1,Y1,X2,Y2),
         BGImage->Canvas,Rect(X1,Y1,X2,Y2));
}
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::ClearSmallBlock(int X, int Y)
{
        int X1,X2,Y1,Y2;
        X1 = X*SmallWidth;
        X2 = X1+SmallWidth;
        Y1 = Y*SmallHeight;
        Y2 = Y1+SmallHeight;
        NextImage->Canvas->Brush->Color = clBlack;
        NextImage->Canvas->FillRect(Rect(X1,Y1,X2,Y2));
}
//---------------------------------------------------------------------------
```

两个擦除函数略微有些不同，对应于游戏容器的擦除函数是将背景的正确部分绘制在指定区域（假设 BGImage 组件已经装载了背景图像），对应于预览框的擦除函数只对指定区域进行黑色填充，因为预览框的背景固定为黑色。

2. 显示方块

显示方块函数 WriteObject 用于将方块从旧位置移动到新位置。

```
void __fastcall TTetrisForm::WriteObject()
{
        if(MoveFlag)
                for (int i=0;i<4;i++)
                        ClearBlock(OldX[i],OldY[i]);
        MoveFlag=true;
        for (int i=0;i<4;i++){
                OldX[i]=ObjectX+Style[Patn][Rotat*10+i*2]-1;
```

```
            OldY[i]=ObjectY+Style[Patn][Rotat*10+i*2+1]-1;
            DrawBlock(OldX[i],OldY[i],Colors[ObjectColor]);
        }
    }
```

MoveFlag 标志记录是第一次显示方块，还是移动后显示方块。对 MoveFlag 进行判断，如果是移动显示则首先擦除原位置的方块（方块的四个小块）。

在指定位置绘制方块（四个小块），并将绘制位置记录在 OldX 和 OldY 数组中，用于下一次擦除。

3. 生成并显示下一个方块

生成并显示下一个方块的函数是 WriteNext。

```
void TTetrisForm::WriteNext()
{
    NextPatn=floor(random(21));
    NextRotat = floor(random(4));
    NextColor = floor(random(7));
    for (int i=0;i<4;i++)
        for (int j=0;j<4;j++)
            ClearSmallBlock(i,j);
    for (int i=0;i<4;i++)
        DrawSmallBlock(Style[NextPatn][NextRotat*10+i*2]-1,
    Style[NextPatn][NextRotat*10+i*2+1]-1,
    Colors[NextColor]);
}
```

采用随机的方式产生下一个方块的样式、旋转状态和颜色，随后清除预览框，并调用 DrawSmallBlock 函数绘制下一个方块的四个小块。

4. 方块动作函数

当方块要改变目前状态而发生某个动作时，这个动作不一定可以被接受。比如方块的某一部分超出了容器，或者与已有方块发生了碰撞。方块动作函数包括 MoveLeft、MoveRight、MoveDown 等，其主要任务是检测方块动作是否可以被接受。

Source 方块动作函数的实现

```
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::MoveLeft()
{
    // 方块左移函数
    int xx=Style[Patn][Rotat*10]+ObjectX-2;
    if (xx >=0){
```

```
                xx=Style[Patn][Rotat*10+2]+ObjectX-2;
                if (xx >=0){
                        xx=Style[Patn][Rotat*10+4]+ObjectX-2;
                        if (xx >=0){
                        xx=Style[Patn][Rotat*10+6]+ObjectX-2;
                        if (xx >=0){
                        if (HitCheck(ObjectX-1,ObjectY,Rotat)){
                                ObjectX--;
                                WriteObject();
                        }
                        }
                        }
                }
        }
}
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::MoveRight()
{
        // 方块右移函数
        int xx=Style[Patn][Rotat*10]+ObjectX;
        if (xx <10){
                xx=Style[Patn][Rotat*10+2]+ObjectX;
                if (xx <10){
                        xx=Style[Patn][Rotat*10+4]+ObjectX;
                        if (xx <10){
                        xx=Style[Patn][Rotat*10+6]+ObjectX;
                        if (xx <10){
                        if (HitCheck(ObjectX+1,ObjectY,Rotat)){
                                ObjectX++;
                                WriteObject();
                        }
                        }
                        }
                }
        }
}
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::Rotate()
{
        // 方块旋转动作函数
```

```
    int Ro=(Rotat+1)%4;
    int xx,yy;
    for (int i=0;i<4;i++){
        xx=ObjectX+Style[Patn][Ro*10+i*2]-1;
        yy=ObjectY+Style[Patn][Ro*10+i*2+1]-1;
        if (xx>9||xx<0||yy>22||yy<0) return;
    }
    if (HitCheck(ObjectX,ObjectY,Ro)){
        Rotat=Ro;
        WriteObject();
    }
}
//---------------------------------------------------------------------------
```

在方块的左移、右移和旋转函数判断了方块（四个小块）是否超出边界，检测是否与原有方块发生碰撞的函数是 HitCheck，此函数通过三个参数传递新的横纵坐标和旋转状态。

```
bool __fastcall TTetrisForm::HitCheck(int X, int Y,int Ro)
{
    int xx,yy;
    for (int i=0;i<4;i++){
        xx=X+Style[Patn][Ro*10+i*2]-1;
        yy=Y+Style[Patn][Ro*10+i*2+1]-1;
        if(GlassMap[xx][yy]!=0) return false;
    }
    return true;
}
```

方块下移时需要做的工作较多，这在前文的分析中已经提到。如果下移失败，则表示方块已经沉底，需要进行一系列处理并准备下一个方块的初始化工作。

方块下移动作函数为 MoveDown。

Source 方块下移函数，包括方块落底的判断、消行处理

```
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::MoveDown()
{
    bool chk=false;
    int xx,yy=Style[Patn][Rotat*10+1]+ObjectY;
    // 判断方块能否下移
    if (yy<23){
    yy=Style[Patn][Rotat*10+3]+ObjectY;
    if (yy<23){
```

```
 yy=Style[Patn][Rotat*10+5]+ObjectY;
if (yy<23){
yy=Style[Patn][Rotat*10+7]+ObjectY;
if (yy<23){
        if (chk=HitCheck(ObjectX,ObjectY+1,Rotat)){
                ObjectY++;
                WriteObject();
        }
}
}
}
}
if (!chk){ // 继续下移失败，方块沉底
        Score+=25;
        // 方块放下音效
        if(PlaySnd) sndPlaySound("Tick.wav",SND_ASYNC);
        // 存储方块数据
        for (int i=0;i<4;i++){
                xx=ObjectX+Style[Patn][Rotat*10+i*2]-1;
                yy=ObjectY+Style[Patn][Rotat*10+i*2+1]-1;
                GlassMap[xx][yy]=ObjectColor+1;
        }
        ScanLines(); // 消行处理
        // 级别提升处理。
        if (Score>4000)  Level = 2;
        if (Score>8000)  Level = 3;
        if (Score>13000) Level = 4;
        if (Score>20000) Level = 5;
        if (Score>30000) Level = 6;
        if (Score>50000) Level = 7;
        Timer1->Interval = 800-Level*100;        // 升级后改变方块自动下落速度。
        if (Level!=StrToInt(Label3->Caption)&&PlaySnd)
                sndPlaySound("Upgrade.wav",SND_ASYNC); // 升级音效
        Label3->Caption=AnsiString(Level);
        Label4->Caption=AnsiString(Score);
        // 准备并显示下一个方块
        ObjectX=3;
        ObjectY=0;
        Patn=NextPatn;
        Rotat=NextRotat;
```

```
            ObjectColor=NextColor;
            MoveFlag=false;
            WriteObject();
            if(!HitCheck(ObjectX,ObjectY,Rotat)){
                // 如果下一个方块填满了容器
                Timer1->Enabled=false;
                isPlaying=false;
                MessageBox(Handle,"抱歉！ 您失败了……","游戏结束",
                    MB_ICONINFORMATION|MB_OK);
                MediaPlayer1->Stop();
                AddHighScore();        // 高分纪录处理
                Button1->Enabled=true;
                N2->Enabled=true;
                Button2->Enabled=false;
                Button3->Enabled=false;
            }
            WriteNext(); // 生成并在预览框显示下一个方块
        }
}
//---------------------------------------------------------------------------
```

方块下移动作的逻辑结构是很清楚的，其算法的建立基于对游戏特点和进行方式的深入分析。

5. 消行处理

消行处理函数中需要扫描 GlassMap 数组。如果发现消行，则需要重置 GlassMap 数组并重新显示游戏容器。

Source 消行处理函数 ScanLines

```
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::ScanLines()
{
    int Lines=0;
    for (int i=0;i<GlassHeight;i++)
    for (int j=0;j<GlassWidth;j++)
        OldGlassMap[j][i]=GlassMap[j][i]; // 保存消行前容器状态
    for (int i= 0;i<GlassHeight;i++)
    {       // 扫描 GlassMap，寻找满行
        int k= 0;
        for (int j=0;j<GlassWidth;j++)
```

```
            if (GlassMap[j][i]>0) k++;
            if (k==GlassWidth){
                Lines++;
                // 在 GlassMap 中消去满行
                for (int a=i;a>0;a--)
                    for (int b=0;b<GlassWidth;b++)
                        GlassMap[b][a]= GlassMap[b][a-1];
            }
        }
        // 重画游戏容器
        for (int i=0;i<GlassHeight;i++)
            for (int j=0;j<GlassWidth;j++)
                if (OldGlassMap[j][i]!=GlassMap[j][i])
                    if (GlassMap[j][i]==0) ClearBlock(j,i);
                    else DrawBlock(j,i,Colors[GlassMap[j][i]-1]);
        // 消行加分
        switch(Lines){
        case 1: Score +=100;
            break;
        case 2: Score +=250;
            break;
        case 3: Score +=550;
            break;
        case 4: Score +=1200;
            break;
        }
        // 消行音效
        if (Lines>0&&PlaySnd) sndPlaySound("Line.wav",SND_ASYNC);
    }
    //---------------------------------------------------------------------------
```

6. 使程序运作起来

以上实现了程序中的主要模块。我们知道，Windows 程序以事件方式驱动，所以需要通过事件使各个模块相互联系并协同工作。

Start 函数在用户单击“开始游戏”菜单或“开始”按钮时执行，开始俄罗斯方块游戏的前期工作。

```
void __fastcall TTetrisForm::Start()
{
    if(PlayMusic)
        MediaPlayer1->Play();
```

```
        // 重新绘制游戏容器
    for (int i=0;i<GlassWidth;i++)
        for (int j=0;j<GlassHeight;j++){
            GlassMap[i][j]=0;
            ClearBlock(i,j);
        }
    // 初始化状态变量
    Level = 1;
    Score = 0;
    Label3->Caption="1";
    Label4->Caption="0";
    MoveFlag=false;
    ObjectX=3; ObjectY=0;
    // 生成并显示一个新的方块
    WriteNext();
    Patn = NextPatn;
    ObjectColor = NextColor;
    Rotat=NextRotat;
    WriteObject();
    // 生成下一个方块
    WriteNext();
    isPlaying=true; // 键盘事件打开
    // 游戏开始。定时器打开。
    Timer1->Interval=800-Level*100;
    Timer1->Enabled=true;
}
```

通过 Start 函数，使定时器和键盘事件函数可以工作，由定时器事件和键盘事件驱动整个游戏。

定时器事件执行方块的是 MoveDown 动作。

```
void __fastcall TTetrisForm::Timer1Timer(TObject *Sender)
{
    MoveDown();
}
```

键盘事件判断用户按键并执行相应的方块动作。

```
void __fastcall TTetrisForm::FormKeyDown(TObject *Sender, WORD &Key,
    TShiftState Shift)
{
    if (!isPlaying) return;
    switch(Key){
    case VK_LEFT:
```

```
            MoveLeft();
            break;
        case VK_RIGHT:
            MoveRight();
            break;
        case VK_UP:
            Rotate();
            break;
        case VK_DOWN:
            MoveDown();
            break;
    }
}
```

可以看到，合理的构建模块使得程序运作的结构非常清晰。

至此俄罗斯方块游戏的核心部分已经完成，它已经可以出色的运转起来了（当然，还应该完成一些细节工作）。优秀的规划和设计使得具体代码的实现变得合理而有序。

8.3.2　其他问题

程序的主要代码部分已经在上一小节完成，现在对程序中其他一些问题加以简单的说明。

1. 图像背景

将图像装入 BGImage 图像组件的是函数 LoadImage。

```
void __fastcall TTetrisForm::LoadImage(AnsiString FileName)
{
    TImage *tmpImage=new TImage(this);
    tmpImage->AutoSize=true;
    tmpImage->Picture->LoadFromFile(FileName);
    if (tmpImage->Width>Image->Width&&tmpImage->Height>Image->Height)
        BGImage->Canvas->Draw(0,0,tmpImage->Picture->Graphic);
    else
        BGImage->Canvas->StretchDraw(
            Rect(0,0,Image->Width,Image->Height),
            tmpImage->Picture->Graphic);
    for (int i=0;i<GlassWidth;i++)
        for (int j=0;j<GlassHeight;j++)
            ClearBlock(i,j);
    delete tmpImage;
```

```
}
```

如果图像尺寸大于需要尺寸时直接装入，如尺寸不合适则作简单拉伸后装入，也可以做到按比例拉伸，这在前面几章中已多次应用。

装入图像后使用 ClearBlock 函数将背景显示出来，注意在本程序中，只允许在游戏没有开始前更改背景，所以上述算法没有问题。但也可以做一些改变，使更改背景功能工作得更好。

用户设置背景图像的响应函数是 N2Click。

```
void __fastcall TTetrisForm::N2Click(TObject *Sender)
{
        if(OpenPictureDialog1->Execute()){
                ImageFileName=OpenPictureDialog1->FileName;
                LoadImage(ImageFileName);
        }
}
```

这里用到了 TOpenPictureDialog 组件，它与 TOpenDialog 组件用法相同，但支持图像预览功能，其效果如图 8-3 所示。

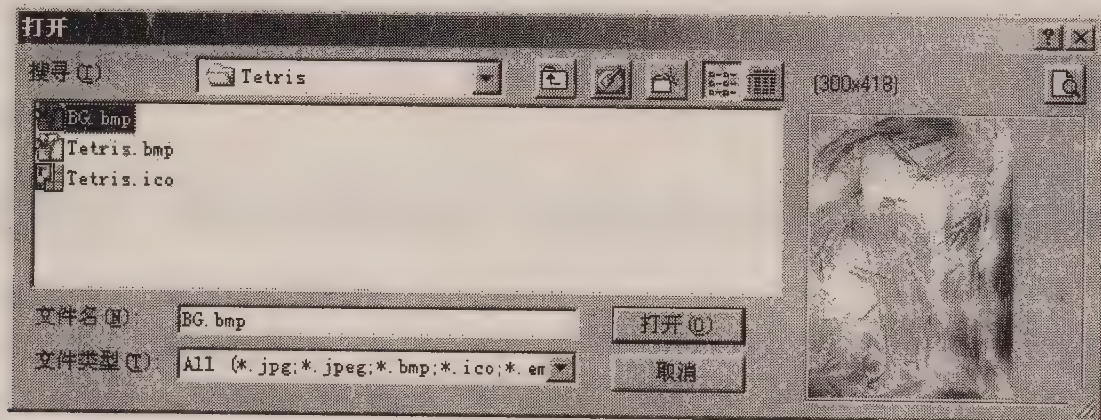


图 8-3　使用 TOpenPictureDialog 组件效果图

2. 播放音乐

只需注意响应 TMediaPlayer 组件的 OnNotify 事件，就可以使背景音乐连续播放。

OnNotify 事件响应函数如下：

```
void __fastcall TTetrisForm::MediaPlayer1Notify(TObject *Sender)
{
        if (MediaPlayer1->NotifyValue==nvSuccessful)
                MediaPlayer1->Play();
}
```

8.4　实用技巧

Windows 应用程序在某种意义上是风格统一的，标准的 Win32 应用程序具备某些通用的

特性或风格，例如帮助系统、INI 初始化文件以及溅出屏幕 SplashScreen 等。

在前几章中完成的实例都是真正的 Win32 应用程序，但我们忽略了作为完整的 Win32 应用程序应该具备的一些特性或风格。最明显的就是没有为任何一个程序创建帮助系统。

本节的任务是讨论包括帮助系统、INI 配置信息文件、溅出屏幕几个 Windows 编程的实用技术，并在“俄罗斯方块”游戏程序中实践它们。

8.4.1 创建帮助系统

对于专业的应用程序来说，联机帮助是程序不可获缺的部分，提供风格统一的帮助系统是 Windows 应用程序的一个重要特点。本小节将讨论 Windows 帮助的技术问题，设计和建立联机帮助的步骤，以及在 C++ Builder 编程中如何为应用程序加入帮助文件。

Windows 提供了操作系统级的联机帮助支持（包括帮助文件格式和 WinHelp API）。Windows 帮助系统的最终是以扩展名为 HLP 的文件的形式出现的。HLP 文件是一种超文本格式文件，并以主题为主线进行编写的，一个主题可以跳转至相关的主题，也可按关键字进行主题查询。帮助文件与软件开发工具相结合，可实现应用程序与帮助文件相应内容的关联。例如，在“俄罗斯方块”程序中的帮助文件系统如图 8-4 所示。

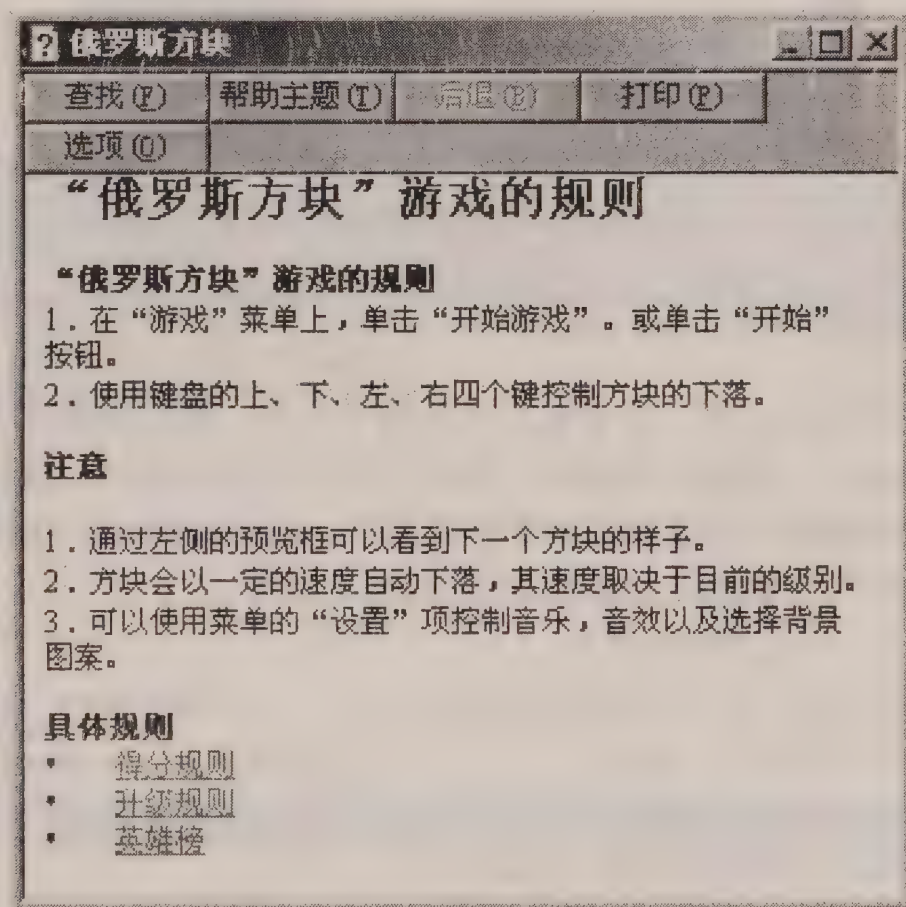


图 8-4 “俄罗斯方块”程序的帮助

除了使用第三方的帮助文件制作工具之外，创制帮助文件的最一般的方法是使用某种可以生成 RTF 格式文本的字处理器，加上 Microsoft Help Workshop 帮助文件建立工具，如图 8-5 所示。下面将简要介绍使用这种方法所需的基本步骤。

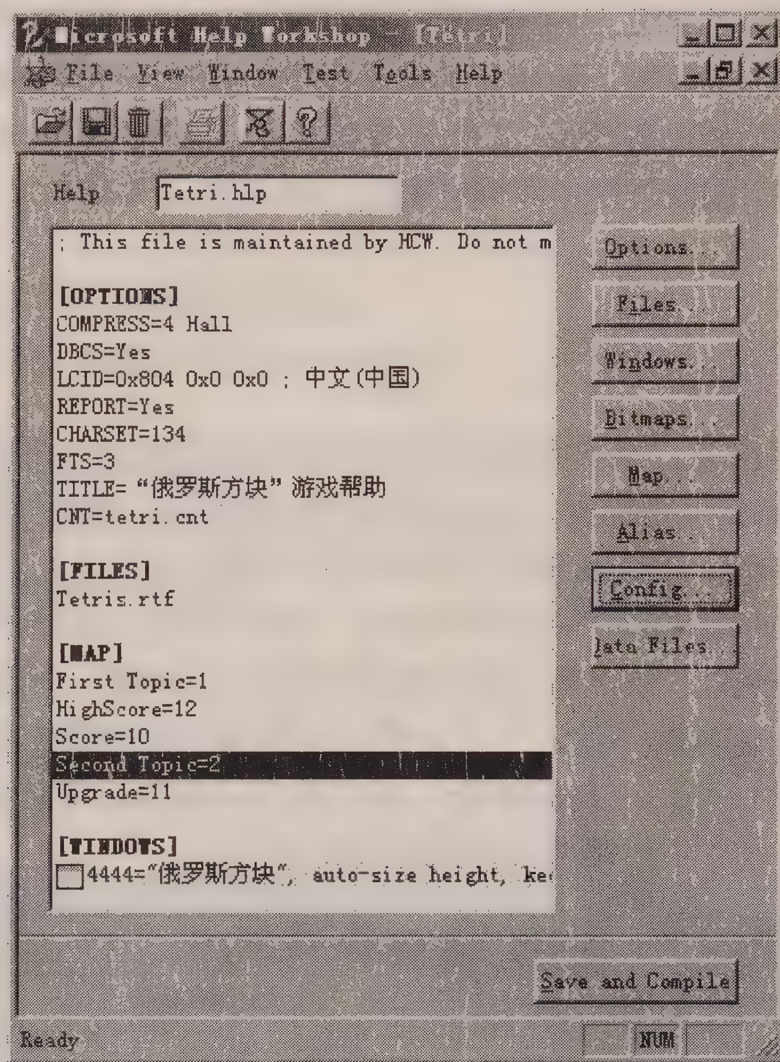


图 8-5　应用 Microsoft Help Workshop 工具建立帮助

1. 制作帮助文件

在最终的帮助文件出炉之前，需要建立三个相关文件：帮助系统文本（RTF 格式）文件、内容文件（Content）、帮助工程文件。拥有了这三个文件之后，就可以利用 Microsoft Help Workshop 帮助编译器将它们组合成 HLP 格式文件。C++ Builder 5 中附带了 Microsoft Help Workshop 工具，它位于 C++ Builder 目录的 Help\Tools\目录下。

建立一个帮助文件，可以归纳为四步。

(1)　建立帮助文本文件，并以 RTF 格式保存。

(2)　建立帮助内容（*.CNT）文件。

(3)　建立帮助工程（*.HPJ）文件。

(4)　将工程文件编译成帮助源文件（*.HLP）。

2. 建立帮助文本文件

建立帮助文本文件可以使用任何一种完全支持 RTF 多文本格式的字处理器，常见的 Microsoft Word 工具就完全可以满足要求。

前面说过，帮助文件是以主题组织的，帮助主题包括主题题目（Title）、主题文本（Text）、脚标和主题内容。主题题目写在主题的第一行，一般使用不同的字体大小、颜色以示区别。

写完题目后，可输入主题的文本，输入时不用担心每行的宽度。编译好的帮助文件会根据窗口大小自动确定行宽。在每个主题的最后插入一个强制分页符（插入\分隔符\分页符），Help Workshop 把每页视为一个单独主题。

在每一个主题的最前端插入#脚标。这时主题文本下端也会出现#，在此后键入内容字符串。WinHelp 使用内容字符串作为唯一的辨识主题，它必须是唯一的，而且不能含有空格和标点符号，它在最终帮助文件中是永远不可见的。

在每一个主题的最前端#脚标后插入$脚标，在文本下端的$脚标处，输入主题的标题，该标题是主题的查询文本字符串，通常与第一行出现的主题题目一致。用户将在“查找”对话框、历史列表，以及书签菜单中看到该字符串。

在每一个主题的最前端$脚标后插入 K 脚标，在主题文本中的 K 脚标后输入该主题的关键字字符串，这些字符串将出现在索引列表框中。需要说明的是，可以为一个主题提供多个主关键字，这些关键字由分号隔开。同时，也可以为索引中主关键字提供副关键字，这些关键字由逗号隔开。

在主题文本中与其他帮助页链接的字段，应该将其字体格式化为双下划线。并在其后紧随一个辨识主题字符串（#脚标所定义的），并将这个字符串字体格式化为隐藏文本。

在主题文本中与其他弹出式帮助页链接的字段，应该将其字体格式化为单下划线，在其后紧随一个辨识主题字符串（#脚标所定义的），并将这个字符串字体格式化为隐藏文本。

经过上述的工作，现在应该已经完成了帮助文本的 RTF 格式文件。图 8-6 显示了“俄罗斯方块”程序中的帮助文本文件，它包含了多个简单的主题，并包含了一些链接。

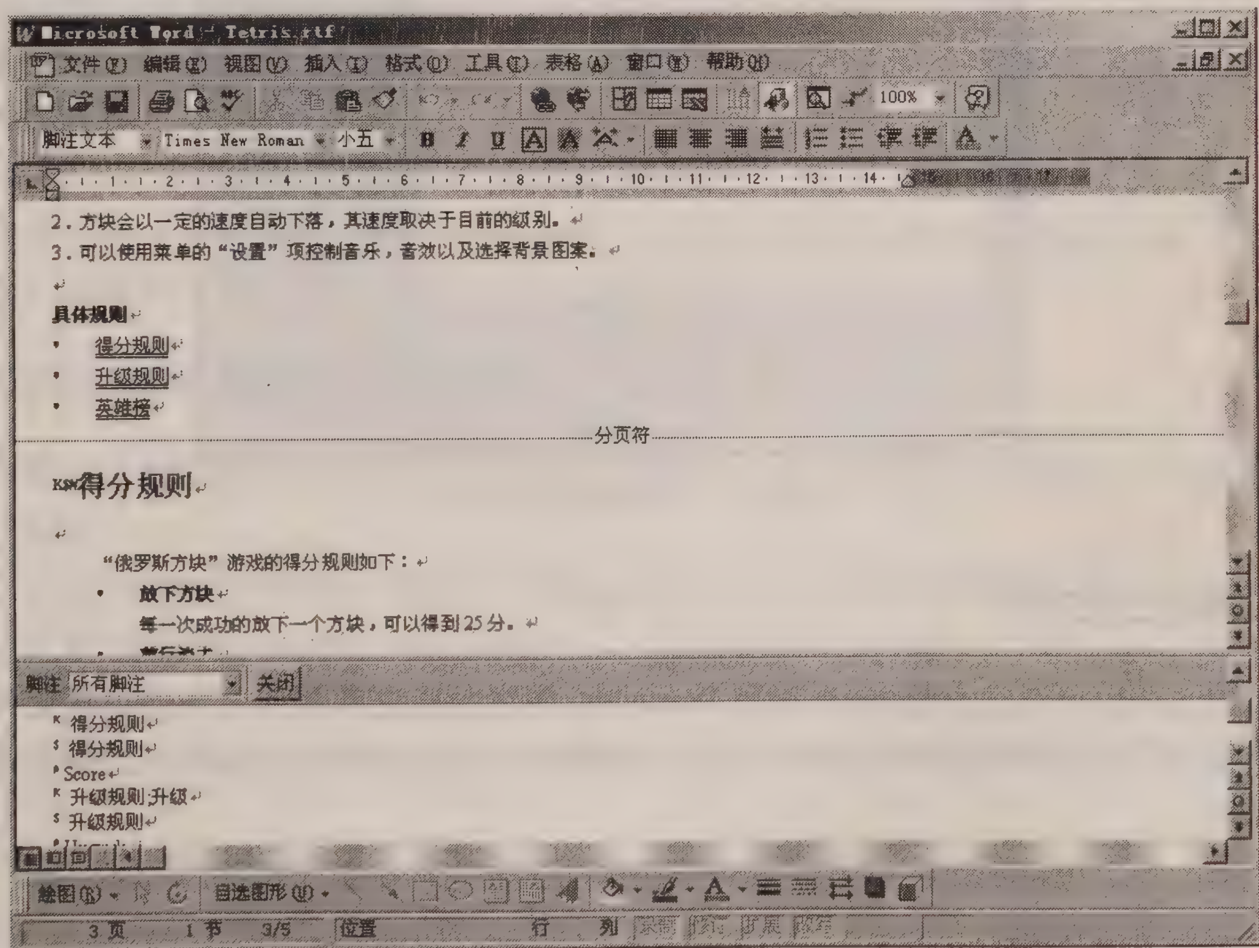


图 8-6　在 Word 中建立帮助文本文件

3. 建立帮助内容文件

帮助内容文件可以在 Microsoft Help Workshop 中直接建立和制作，如图 8-7 所示，在 Help Workshop 中建立帮助内容文件可以输入与之相连的 HLP 帮助文件、帮助文件的缺省窗口类型（在帮助工程文件中定义）以及显示在帮助窗口顶端的缺省标题。然后可以在下方的列表框中建立主标题（Heading）和主题（Topic），以及宏（Macro）和包含（Include）。主题与某一个确定的帮助页相连，由帮助页的辨识主题字符串标识（输入在 Topic ID 框中）。

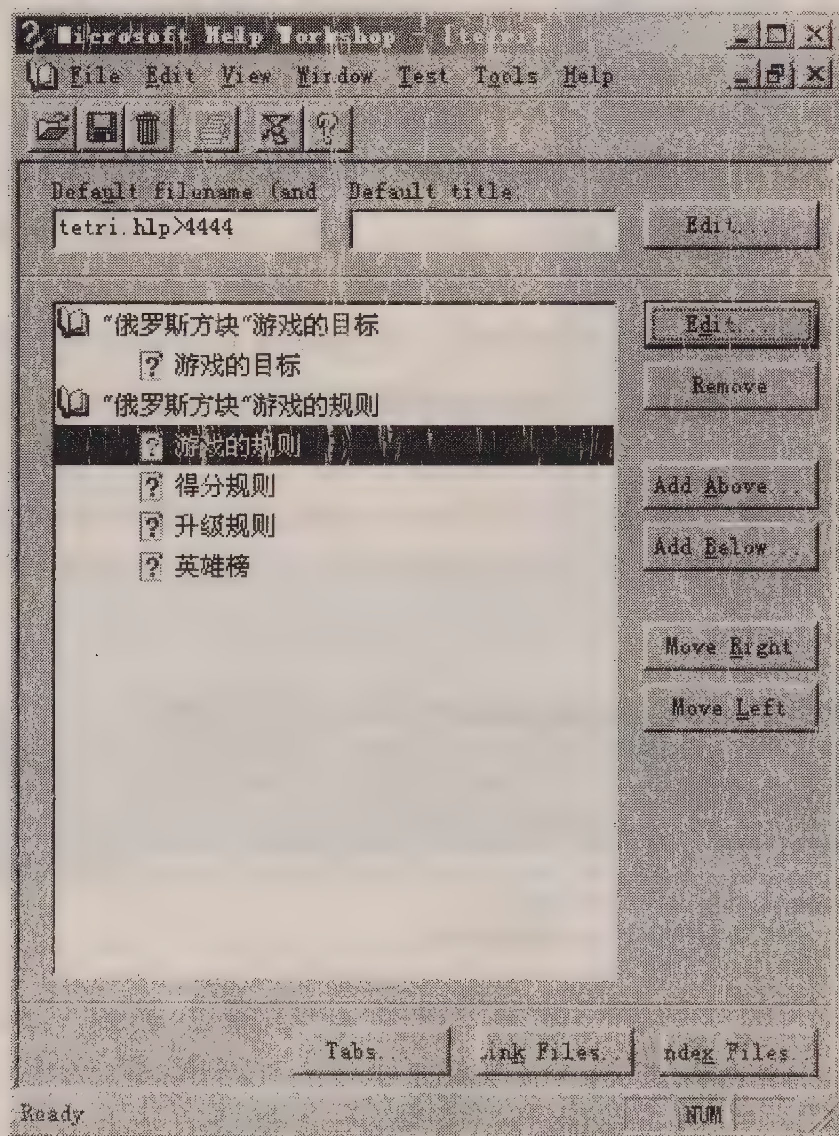


图 8-7　建立帮助内容文件

4. 建立帮助工程文件

帮助工程文件也可以在 Microsoft Help Workshop 中直接建立和制作。帮助工程文件是一个文本文件，包含了有关帮助文件的许多信息，编译器对工程文件进行编译。工程文件的扩展名必须是 HPJ。

一般来说，关于帮助系统的以下信息必须在帮助工程文件中给予确定：

- 帮助文本文件的文件名。
- 帮助内容文件的文件名。
- 压缩技术选项。
- 辨识主题字符串与 ID 号的映射关系。
- 定义窗口类型。

- 最终生成帮助文件的 HLP 名称。

下面是“俄罗斯方块”程序中的帮助工程文件。

```
; This file is maintained by HCW. Do not modify this file directly.
[OPTIONS]
COMPRESS=4 Hall
DBCS=Yes
LCID=0x804 0x0 0x0 ; 中文(中国)
REPORT=Yes
CHARSET=134
FTS=3
TITLE= “俄罗斯方块” 游戏帮助
CNT=tetri.cnt
HLP=Tetri.hlp
[FILES]
Tetris.rtf
[MAP]
First Topic=1
HighScore=12
Score=10
Second Topic=2
Upgrade=11
[WINDOWS]
4444="俄罗斯方块",,61700,(r14876671),(r12632256),f3
```

在完成帮助工程文件后，就可存盘编译。可以在 Microsoft Help Workshop 中运行帮助文件，观察生成结果。

5. 在应用程序中提供帮助支持

为应用程序提供帮助系统的首要任务是将帮助文件与应用程序相连接。可以通过下面的步骤得以实现：

(1) 在 C++ Builder 环境中菜单的【**Project/Option**】选项指定帮助文件名。

(2) 设置主窗体的 HelpContext 属性为 1，用以指向帮助主题。

(3) 设置程序中其他组件或窗体的 HelpContext 属性为相应值。

此外如果需要在菜单中或者按钮上单击打开菜单，则应该使用 TApplication 类提供的相关属性和方法。

- HelpFile 属性。

声明：__property System::AnsiString HelpFile = {read=FHelpFile, write=FHelpFile};

指明应用程序的帮助文件名。

- HelpCommand 方法。

声明：bool __fastcall HelpCommand(int Command, int Data);

提供对任何一种 WinHelp API 中的帮助命令支持。

- HelpContext 方法。

声明：bool __fastcall HelpContext(Classes::THelpContext Context);

显示一个指定的帮助主题。

- HelpJump 方法。

声明：bool __fastcall HelpJump(const System::AnsiString JumpID);

显示一个指定的帮助主题，通过 ID 号标识。

具体到“俄罗斯方块”程序，单击“帮助”按钮使帮助文件打开并显示主题，其响应函数为：

```
void __fastcall TTetrisForm::Button5Click(TObject *Sender)
{
    Application->HelpCommand(HELP_CONTENTS, 0);
}
```

8.4.2 使用 INI 文件

通常许多 Windows 应用程序都需要保存程序的状态信息，以便下次程序执行时恢复原有状态。对于标准的 Win32 应用程序来说，完成上述问题有两个通用途径：使用注册表（Registry）或使用 INI 初始化文件。

有关利用注册表的内容我们已在第 6 章中作了介绍。相比之下，注册表的保密性能较好，但它没有 INI 初始化文件那样易于控制。所以，对于一些小型的或简单的 Windows 应用程序（例如 Windows 的扫雷游戏等）来说，使用 INI 初始化文件是一个更为不错的选择。

INI 类型的文件是以部分（Section）——值（Value）的方式组织的文本文件，例如俄罗斯方块游戏的 INI 文件内容如下：

```
[Option]
Sound=1
Music=1
BGImage=D:\Tetris\Tetris.bmp
[HighScore]
HighScore0=16125
HighScore1=8550
HighScore2=8075
[HighName]
HighName0=小明
HighName1=小明
HighName2=小明
```

方括号内的是部分名称，等号前的是值名称（键名），等号后是具体的值。

1. TIniFile 类

C++ Builder 中有一个 TIniFile 类封装了对 Windows 初始化文件的操作。实例化一个

TIniFile 类就建立了和某个 INI 文件的连接，而后使用 TIniFile 类方法就可以向该文件读取或写入信息。

- EraseSection。

声明：virtual void __fastcall EraseSection(const System::AnsiString Section);

删除 INI 文件的整个一个部分。

- ReadSection。

声明：virtual void __fastcall ReadSection(const System::AnsiString Section, Classes:: TStrings* Strings);

读取 INI 文件的某个指定部分的所有键名到一个 Tstrings 对象。如 TstringList 类，TlistBox 的 Items 属性等等。

- ReadSections。

声明：virtual void __fastcall ReadSections(Classes::TStrings* Strings);

读取 INI 文件所有部分的名称到一个 Tstrings 对象。

- ReadSectionValues。

声明：virtual void __fastcall ReadSectionValues(const System::AnsiString Section, Classes::TStrings* Strings);

读取 INI 文件中某个指定部分内的所有值到一个 TStrings 对象。

- ValueExists 方法。

声明：__fastcall TCustomIniFile(const AnsiString FileName);

确定 INI 文件中指定名称的值是否存在。类似的方法有 SectionExists，用于确定指定名称的部分是否存在。

- WriteInteger 方法。

声明：virtual void __fastcall WriteInteger(const AnsiString Section, const AnsiString Ident, int Value);

在 INI 文件的指定部分写入指定名称的整型值。类似的方法有 WriteBool、WriteDate、WriteDateTime、WriteFloat、WriteString、WriteTime。

- ReadInteger 方法。

声明：virtual int __fastcall ReadInteger(const AnsiString Section, const AnsiString Ident, int Default);

读取 INI 文件的指定部分中指定名称的整型值。如果发生：

- 指定部分不存在。
- 指定值的键名不存在。
- 数据不能被赋值。

则返回 Default 参数的值。类似的方法有：ReadBool、ReadDate、ReadDateTime、ReadFloat、ReadString、ReadTime。

2. 在程序中实现 INI 文件应用

INI 文件的使用较注册表更为方便，使用注册表访问数据时需要 Open 一个主键，而在 INI 文件中，可以直接访问整个文件的任何一项数据。另外，TIniFile 类读取方法的 Default 功能的提供也是一个非常方便的设计。

具体在“俄罗斯方块”程序中，通过在 OnCreate 函数中调用 ReadIni 函数进行 INI 初始化文件的读取工作。

```
void TTetrisForm::ReadIni()
{
    IniFile=new TIniFile(
        ExtractFilePath(Application->ExeName)
        +"\\"+IniFileName);
    PlaySnd=IniFile->ReadBool("Option","Sound",true);
    PlayMusic=IniFile->ReadBool("Option","Music",true);
    ImageFileName=IniFile->ReadString("Option","BGImage","BG.bmp");
    for (int i=0;i<3;i++){
        HighScore[i]=IniFile->ReadInteger("HighScore","HighScore"+IntToStr(i),0);
        HighName[i]=IniFile->ReadString("HighName","HighName"+IntToStr(i),"匿名");
    }
    N4->Checked=PlaySnd;
    N5->Checked=PlayMusic;
}
```

由于 TIniFile 类的读取方法提供 Default 参数，因此不必判断部分或值是否存在。

在 OnClose 响应函数中，将更新数据写入 INI 文件。

```
void __fastcall TTetrisForm::FormClose(TObject *Sender,
TCloseAction &Action)
{
    IniFile->WriteBool("Option","Sound",PlaySnd);
    IniFile->WriteBool("Option","Music",PlayMusic);
    IniFile->WriteString("Option","BGImage",ImageFileName);
    for (int i=0;i<3;i++){
        IniFile->WriteInteger("HighScore","HighScore"+IntToStr(i),HighScore[i]);
        IniFile->WriteString("HighName","HighName"+IntToStr(i),HighName[i]);
    }
    delete IniFile;
}
```

8.4.3 溅出屏幕（Splash Screen）

溅出屏幕（Splash Screen）是一种在 Windows 应用程序中使用的典型技术。它是一个先于主窗体显示的初始窗口，用于在应用程序进行初始化工作时显示。一般来说，溅出屏幕被处理为一个漂亮的图像，它不仅可以向用户显示一些信息，同时可以使得应用程序更具有吸引力。图 8-8 展示了“俄罗斯方块程序”的溅出屏幕。



图 8-8 “俄罗斯方块”程序的溅出屏幕

溅出屏幕给用户造成的第一印象是不容忽视的。这方面优秀的例子很多，如 Visual C++ 6，Boland C++ Builder 4 等都有一个漂亮的溅出屏幕。利用 C++ Builder 的强大功能，也可轻易地为自己的应用程序添加溅出屏幕。

溅出屏幕在技术上有一个小小的问题，即溅出屏幕也是一个通常的窗体，显示方法也与通常的显示方法无异（TForm 类的 Show 方法），但是它必须先于主窗体显示。所以关于溅出屏幕的创建，调整和显示工作应该在工程的主入口文件 Tetris.cpp 中 WinMain 函数中进行。

还是以“俄罗斯方块”程序为例，创建一个新窗体，窗体上放置一个 Timage 组件，并为此组件装入设计好的图片，将窗体和 Timage 组件的 AutoSize 属性均设为 true，这样可以使得窗体大小和图片原始尺寸一致。

在 Splash 窗体上放置一个 TTimer 定时器组件，调整定时器组件的 Interval 属性为 2s 或 3s。

此外应该将此窗体设为 Available form。

在工程的主入口文件 Tetris.cpp 中修改代码如下：

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        TSplashForm *Splash = new TSplashForm(Application);
        Splash->FormStyle=fsStayOnTop;
        Splash->BorderStyle=bsNone;
        Splash->Position=poScreenCenter;
        Splash->Show();
```

```
            Application->HelpFile = "Tetri.hlp";
            Application->Title = "俄罗斯方块";
            Application->CreateForm(__classid(TTetrisForm), &TetrisForm);
            Application->CreateForm(__classid(TAboutBox), &AboutBox);
            Application->CreateForm(__classid(TScoreForm), &ScoreForm);
            Application->Run();
        }
        catch (Exception &exception)
        {
            Application->ShowException(&exception);
        }
        return 0;
}
```

程序中标注出的代码的作用就是建立并显示一个 Splash 屏幕。FormStyle 设为 fsStayOnTop 使得 Splash 屏幕出现在主窗体之上；BorderStyle 属性设为 bsNone 删除窗体的边框和标题；Position 属性设为 poScreenCenter 使窗体位于屏幕中心。这三点是建立 Splash 屏幕所必需的。

从上可以看出，并没有在这段代码中关闭溅出屏幕。一般来说，可以在主窗体初始化代码执行完毕后关掉 Splash 屏幕，但这个程序中主窗体初始化代码较少，为了能够出现明显的 Splash 屏幕效果，此处借助 Splash 窗体中的定时器事件来关闭。

响应定时器事件的函数为：

```
void __fastcall TSplashForm::Timer1Timer(TObject *Sender)
{
        Close();
        Release();
}
```

这里用到了 TForm 类的 Release 方法，此方法与 Free 方法的功能类似。但使用 Release 方法清除自己只有等到所有的事件处理程序都执行完成后才开始。在窗体内对自身使用 Free 方法是不可以的，这样做会造成访问错误，因为没有结束的事件处理程序可能引用了窗体对象。

8.5 为游戏程序增加手柄支持

对于游戏编程任务来说，往往需要涉及游戏手柄（或称游戏杆）这一特殊的输入设备，使用游戏手柄进行游戏可以使游戏程序的趣味性大增。如何编程为游戏增加对游戏手柄的支持？

在 Windows 编程中，系统直接提供游戏手柄设备的相关支持，Windows 的多媒体 API 中包含了一组检测、控制和使用游戏手柄设备的函数，使用这些函数使得在程序中增加游戏手柄支持变得相当容易。

图 8-9 展示了如何在 Windows 中设置游戏的手柄，从中可以看到，游戏手柄有轴（方向键）和按钮两种不同的输入方式。

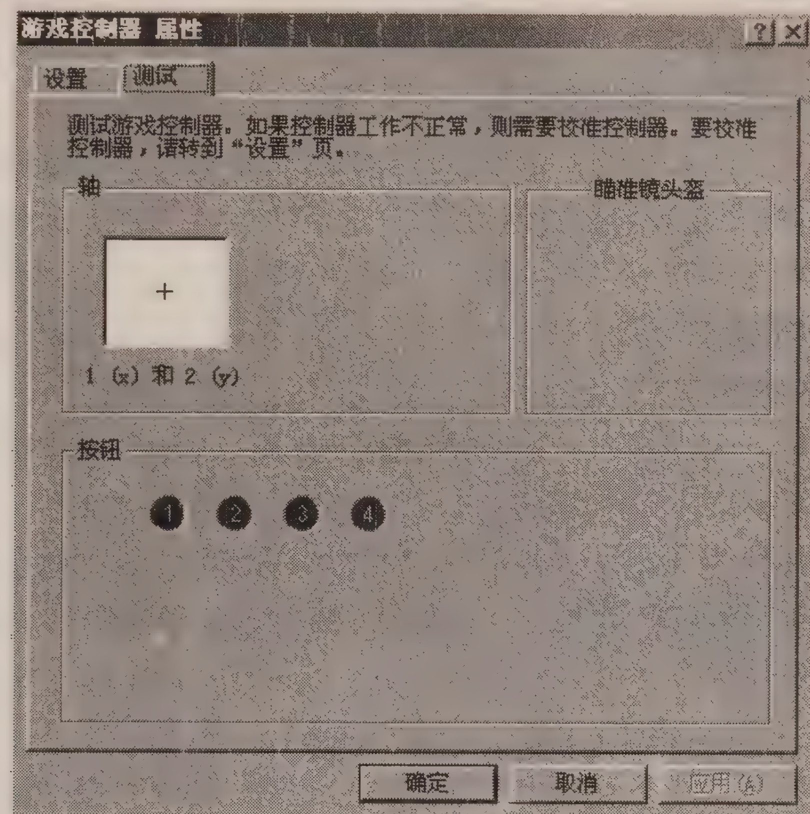


图 8-9　在 Windows 中设置游戏手柄

游戏手柄和其他输入设备（如鼠标、键盘）一样，在 Windows 中是通过消息来反馈并进行控制的。但 C++ Builder 中并没有与游戏杆相关的事件，所以在应用程序中，必须自己映射游戏杆消息，包括按键消息和轴（方向键）移动消息。

为了在游戏中启用游戏手柄，首先要检测用户的计算机中是否正确安装了游戏手柄，这包括检查驱动程序支持和确认游戏手柄已和系统正确相连两项工作。检测系统端口和驱动程序支持需要调用 joyGetNumDevs 函数，如果结果为 0，则表示系统不支持游戏手柄。但即使返回值不为 0（一般来说，正常返回值为 16），也不能确定游戏杆已和系统正确连结了，必须通过 joyGetPosEx 进行进一步确认，并检测是否有错误发生。

如果已经确定用户的计算机有游戏杆可用，就可以接受手柄发来的消息。这需要调用 Windows API 函数 joySetCapture，该函数声明如下：

```
MMRESULT joySetCapture(
  HWND hwnd,        // 接收方 Windows 句柄
  UINT uJoyID,      // 游戏手柄设备 ID
  UINT uPeriod,     // 接收频率
  BOOL fChanged     // 位置改变标志
 );
```

这个函数的第一个参数是要接受的应用程序窗口句柄，第二个参数指定需要接收消息的游戏手柄设备 ID。uPeriod 是一个重要参数，它指定以多大的频率接收 MM_JOYxMOVE 消息（方向键消息，x 为操纵杆号），单位是毫秒。这个消息和一般的输入设备消息不同，无论方向键是否移动，都将以这个消息收到 MM_JOYxMOVE 消息。

在“俄罗斯方块”程序中，检测和连结手柄在函数 ConnectJoy 中实现。

Source 检测和连结游戏手柄

```
//---------------------------------------------------------------------------
void __fastcall TTetrisForm::ConnectJoy()
{
        MMRESULT Result;
        JOYINFO JoyInfo;
        if (joyGetNumDevs()==0){
                J1->Enabled=false;
                return;
        }
        Result=joyGetPos(JOYSTICKID1,&JoyInfo);
        if (Result!=JOYERR_NOERROR){
                J1->Enabled=false;
                return;
        }
        if (J1->Checked){
                joyGetDevCaps(JOYSTICKID1,&JoyCaps,sizeof(JOYCAPS));
                joySetCapture(Handle,JOYSTICKID1,2*JoyCaps.wPeriodMin,false);
        }
}
//---------------------------------------------------------------------------
```

然后可以建立消息映射，处理 MM_JOYxMOVE 消息和 MM_JOYxBUTTONDOWN 消息，具体代码如下。

主窗体单元头文件中：

```
BEGIN_MESSAGE_MAP
        MESSAGE_HANDLER(MM_JOY1MOVE,TMessage,JoyMove);
        MESSAGE_HANDLER(MM_JOY1BUTTONDOWN,TMessage,JoyPress);
END_MESSAGE_MAP(TForm)
```

消息处理函数：

```
void __fastcall TTetrisForm::JoyPress(TMessage &msg)
{
        if (!isPlaying) return;
        if (msg.WParam & JOY_BUTTON1)
                Rotate();
}
void __fastcall TTetrisForm::JoyMove(TMessage &msg)
{
        if (!isPlaying) return;
```

```
        int PosX,PosY;
        PosX=msg.LParamLo;
        PosY=msg.LParamHi;
        if (PosX<10000)
            MoveLeft();
        else if (PosX>55000)
            MoveRight();
        if (PosY>55000)
            MoveDown();
}
```

对于 MM_JOYxBUTTONDOWN 消息的响应函数，几个常量（JOY_BUTTONx）和 msg 的 WParam 值按位求与（“&”）的结果表示该按钮是否被按下，在程序中判断手柄上的第一个按钮是否被按下，该按键执行方块的翻转。

对于 MM_JOYxMOVE 消息的响应函数，msg 的 LParam 参数的高位和低位分别表示方向键的 X 向位置和 Y 向位置。这是一个抽象位置，X 向位置和 Y 向位置的范围都在 0 ~ 65535 之间，当方向键没有按下时，该位置在（32367,32367）附近，可以设定一个位置范围判定是否按下某方向。例如在本程序定义 PosX 值在 0 ~ 10000 之间时判定左方向被按下。

另外，在调用 joySetCapture 函数时可以设置第四个参数为 true，这样将支持当方向轴移动一定距离才接收消息，这个距离值可以由 joySetThreshold 函数设置。

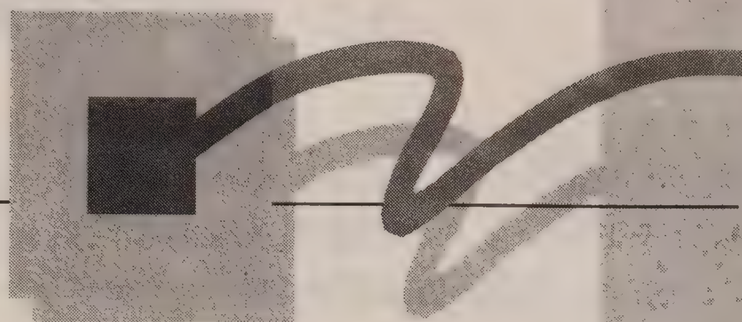
最后，在 OnDesotroy 函数中释放游戏手柄设备：

```
void __fastcall TTetrisForm::FormDestroy(TObject *Sender)
{
    if (J1->Checked)
        joyReleaseCapture(JOYSTICKID1);
}
```

第9章

快速文件处理工具——进程和多线程技术

创建和使用进程
在 C++ Builder 中使用多线程
多线程技术的实际应用
优先级和线程通信
线程计时和线程间同步



本章导读

多线程程序能够更好地表述和解决许多现实的编程问题，是计算机引用开发和程序设计的一个必然发展趋势。在本章中，读者将看到 C++ Builder 是如何实现多线程这一 Win32 特色技术的。C++ Builder 的出色封装使开发人员能够快速明了地实现这一点。多线式文件处理工具是本章将要开发的实用工具，它包括快速文件查找和快速文件拷贝两个模块，完全利用了 Win32 的多线程技术。

本章内容包括:

- 进程部分。关于 Win32 中创建和使用进程方法的阐述，实现 CoolARJ 压缩程序的范例。
- 关于 Win32 线程技术原理，以及 C++ Builder 中使用线程的一般讨论。
- 多线程技术应用——多线式文件处理工具程序的实现，具体完成一个应用多线程技术的实例。通过程序的建立详细阐述关于多线程的创建、使用、控制等问题。涉及了程序中使用多线程的几个方面要点：包括用 Synchronize 方法与 VCL 同步、优先级设定和调度、TEvent 和线程通信等。
- 有关多线程处理的高级技术。涉及对线程计时，线程的本地存储，线程间同步等深层问题。

本章内容属于 Windows 9X 和 Windows NT 编程的高级话题，读者将从中掌握 Win32 的强大功能，并步入 Windows 编程的更高阶段。

9.1 进程和进程创建

程序是一段静态的代码，它是应用执行的蓝本。进程则是程序的一次动态执行过程，作为执行蓝本的程序可以被多次加载到系统的不同内存区域，而形成不同进程。本节将讨论进程技术和关于进程技术编程方面的内容。

9.1.1 进程存储

“进程”是指当前载入程序的一个术语。在磁盘中的可执行文件仅仅是一个文件，然而它一旦开始运行，就成为一个进程。

在 Windows 操作系统中，每个进程都有一个与之相对应的 HInstance，这个变量在程序启动时自动生成，应用程序的 HInstance 是程序载入时的基地址。在 Windows 95/98 中，它的典型值是 10X00400000，即 4194304。这表明在 Windows 95/98 中应用程序一般装载在 4MB 的地方。它的范围可以从 4MB 到 2GB。这就意味着 Windows 95 程序有 2GB 减去 4MB 空间可以伸展。程序的所有代码和数据都存在这 2GB 的空间中。也就是说，Win32 操作系统为用户的应用程序提供的几乎无限的虚拟资源（虽然，可能由于硬件的原因而不能完全达到）。

C++ Builder 中，可以轻松的控制在什么地方载入应用程序。这是在 C++ Builder IDE 的菜单中选择【**Project/Options**】的 Linker 页，通过修改 Image Base 项完成。

一个 Win32 的进程可以直接访问多达 2GB 的空间，并且还有另外的 2GB 空间为操作系统所保留。这样意味着一个应用程序被载入了一个 4GB 的内存空间，如此之大的空间显然是无法通过硬件实现的（目前所能配备的最大内存也不过 1GB 左右）。事实上，每个程序所分配的地址并不是 RAM 中的物理地址，而是虚拟地址。操作系统实际只是分配给了一个地址范围。当需要访问这段存储空间时，再把物理地址分配给它。这种把物理地址分配给虚拟存储的方法称为“映射”。

一页存储空间一般为 4KB，即 4096 字节。Win32 操作系统不停的把一页一页的虚拟存储空间映射到内存空间。映射过程中，程序的地址看起来没有变化，这是因为 Windows 通过索引表把虚拟地址转化为物理地址。所以在 Win32 程序中，得到的地址决不会直接指向物理存储器，它们都是间接地址。

虚拟存储使进程与进程之间不受干扰。多个同时运行的进程相互无关，这样进程之间相互覆盖和冲突的概率就大大减小。而在传统的操作系统（如 DOS），一个无效地址的操作可能覆盖其他进程的数据而导致崩溃。


Win32 是 Microsoft 公司的注册商标。它是指 32 位 Windows 操作系统，包括 Windows 95、Windows98 和 WindowsNT。Win32 操作系统比起 16 位 Windows 如 Windows 3.1 是一个巨大的飞跃。它拥有许多革命性的优越的特性，譬如本章所述的多线程技术。本章所示的例程无论如何也是无法在 Windows 3.1 下实现的。

9.1.2 进程创建方法

在 Win32 中，为保证多任务之间的独立性，当各个模块可以分开单独执行时，可以使用多进程。创建进程可以用 CreateProcess 函数。

该函数的原型为：

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

CreateProcess 函数是一个 Windows API 函数，它的定义包含在 Windows.h 头文件中。它的参数多而繁杂，下面简要介绍它的各个参数的含义：

lpApplicationName	指向可执行文件名的字符串，注意必须为路径全名。
lpCommandLine	指向将执行的应用程序的命令行的字符串。若 lpApplicationName 和 lpCommandLine 都不为空，则 lpApplicationName 指定执行模块，lpCommandLine 指定命令行，如果任何一个为空，Windows 使用不空的那一个来调用执行应用程序。
lpProcessAttributes	指向一个安全结构，用以说明创建进程的安全属性。
lpThreadAttributes	用以指定创建进程的主线程的安全属性。
bInheritHandles	决定新进程是否从调用进程中继承句柄。如为真，则调用进程中每个可继承的打开句柄。
dwCreationFlags	为新进程的附加标志。
lpEnvironment	指向一个用于新进程的环境块。如果此参数为空，则新进程使用调用进程的环境。
lpCurrentDirectory	指向一个字符串，该字符串为将创建进程指定当前工作驱动器和目录，必须为一全路径名。如果此字符串为空，则新进程使用调用进程一样的当前工作驱动器和工作目录。
lpStartupInfo	指向 STARTUPINFO 结构，该结构用于指定新进程的主窗口启动时如何显示。这个结构的形式如下：

```
typedef struct _STARTUPINFO
  DWORD   cb;              // 本结构大小，sizeof(STARTUPINFO)
```

```
    LPTSTR lpReserved;      //保留字，在调用 CreateProcess 前应置为空
    LPTSTR lpDesktop;       //指向桌面名，只在 Windows NT 下有效
    LPTSTR lpTitle;         //用于设置控制台进程的窗口标题
    DWORD  dwX;             //新建窗口位置的 X 向坐标值
    DWORD  dwY;             //新建窗口位置的 Y 向坐标值
    DWORD  dwXSize;         //新建窗口的尺寸
    DWORD  dwYSize;
    DWORD  dwXCountChars;       //用于设置控制台进程的一行的总字符数
    DWORD  dwYCountChars;       //用于设置控制台进程的一列的总字符数
    DWORD  dwFillAttribute;     //用于设置控制台进程的字符颜色
    DWORD  dwFlags;             //标志位,指定本结构各个成员的使用方式
    WORD   wShowWindow;         //创建新进程的 ShowWindow 的参数
    WORD   cbReserved2;         //保留字段，应为 0
    LPBYTE lpReserved2;         //保留字段，应为 NULL
    HANDLE hStdInput;           //新进程标准输入句柄
    HANDLE hStdOutput;          //新进程标准输出句柄
    HANDLE hStdError;           //新进程标准错误句柄
} STARTUPINFO, *LPSTARTUPINFO;
```

lpProcessInformation 参数指向一个 PROCESS_INFORMATION 结构，该结构用于接受有关新进程的标识信息。PROCESS_INFORMATION 结构形式如下：

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;        //新创建进程的句柄
    HANDLE hThread;         //新创建进程的主线程句柄
    DWORD dwProcessId;      //新创建进程的 ID
    DWORD dwThreadId;       //新创建进程的主线程 ID
} PROCESS_INFORMATION;
```

创建的新进程完全独立于调用进程，但如果用 CreateProcess 函数创建进程成功，则新进程的用关句柄和 ID 值可以通过 PROCESS_INFORMATION 结构返回，而通过进程句柄则可以获得新创建进程的完全的控制权。

创建进程的另一种选择是使用 WinExec 函数。WinExec 源于 Windows 3.1 操作系统，但在 Win32 中仍然可用。WinExec 函数较 CreateProcess 函数简单得多，但与之相应的是少了很多控制或安全的功能。此函数原型如下：

```
UINT WinExec(
    LPCSTR lpCmdLine,       //命令行
    UINT uCmdShow   //应用程序窗口显示方式
);
```

欲终止一个进程，可以调用 ExitProgress 函数或 TerminateProgress 函数，若使用 ExitProgress，则此进程的所有附属的 DLL 和线程都将被终止。

9.1.3 后台进程：制作 Windows 版的 ARJ 工具

使用多进程的一个最直接的考虑就是将一些为 MS-DOS 环境开发的高性能程序改变成为一个 Windows 界面的程序。当然可以把那些应用程序完全移植，但使用进程创建完成这种工作更加简便和有效。

作为实例，本节将完成名为 CoolARJ 的应用程序。该范例程序利用到创建进程的知识，将 MS-DOS 下的 ARJ 压缩程序改变为漂亮的 Windows 程序，而不留任何痕迹，如图 9-1 所示。

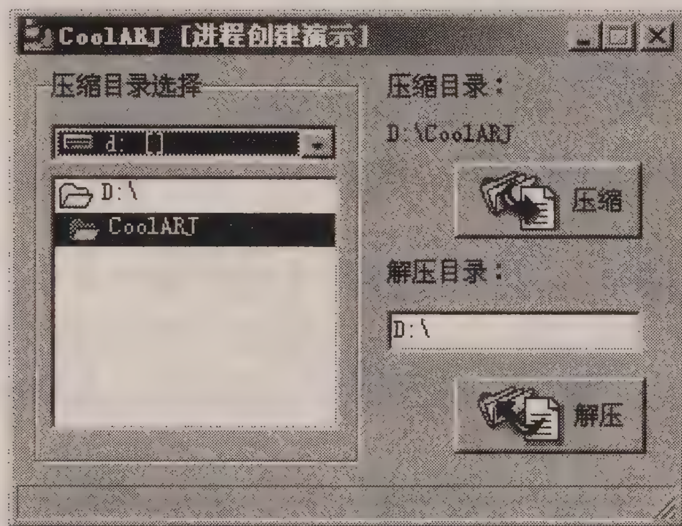


图 9-1 Windows 图形界面的 ARJ 压缩工具——CoolARJ

本程序基于 MS-DOS 下的 ARJ 2.60 版本，实现这个程序的基本想法是由主进程（Windows 界面程序）通过用户选择得到必要的 ARJ 参数，如压缩目录、压缩文件名等等。用户发出命令后，将 DOS 版的 ARJ 程序作为进程调用。调用时不创建窗口，从而使用户感觉不到 DOS 版的 ARJ 程序的运行。这样，压缩或解压过程就在用户不加察觉的情况下完成了。

但仅仅这样还不能称之为“不留痕迹”，因为用户还是能够看到 DOS 版的 ARJ 程序（Arj.exe）在 CoolARJ 程序的目录中存在。这里要作一些巧妙的处理，把 ARJ 程序的 DOS 版本也隐藏起来。

实现这一功能的主要想法是：将 Arj.exe 作为资源编译进 EXE 文件中。在程序运行时，再将 Arj.exe 分离到系统的临时目录下。然后再作为进程调用完成压缩或解压过程。

具体来说，首先建立 CoolARJ.rc 资源文件，如下：

```
CoolARJ.rc:
arjfile EXEFILE arj.exe
```

将 CoolARJ.rc 资源文件加入工程中。而后我们就可以在程序中把 Arj.exe 分离出来并使用了。在本程序中，实现资源分离用到 C++ Builder 的 TResourceStream 类，具体分离过程包含在 FormCreate 响应函数中：

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    char exefile[100], tmppath[100] ;
```

```
        unsigned long ret ;
        GetTempPath(100,tmppath) ;
        strcpy(exefile,(AnsiString(tmppath)+AnsiString("\\arj.exe") ).c_str()) ;
        ret = GetFileAttributes( exefile ) ;
        if( ret == 0xffffffff ) {
            TResourceStream &rs = *new TResourceStream(
                (int)HInstance, AnsiString("arjfile"),"EXEFILE" ) ;
            rs.SaveToFile(AnsiString(exefile)) ;
            delete &rs ;
        }
        SaveDialog1->Filter="ARJ archive files(*.Arj)|*.Arj";
        SaveDialog1->Title="压缩到文件";
        OpenDialog1->Filter="ARJ archive files(*.Arj)|*.Arj";
        OpenDialog1->Title="解压压缩档案";
    }
```

这段代码中，GetTempPath 函数用于获得系统的临时目录。然后判断在临时目录中 Arj.exe 文件是否已经存在，若不存在（ret == 0xffffffff 没有找到文件），则将 Arj.exe 文件分离出来，并用 TResourceStream 类的 SaveToFile 方法保存在系统的临时目录中，以备将来使用。

当用户单击压缩按钮时，将触发 OnClick 事件。在该事件响应函数 BitBtn1Click 中进行压缩文件处理，实现文件压缩归结为进程创建过程：

```
    void __fastcall TForm1::BitBtn1Click(TObject *Sender)
    {
        bool proc_create_state;
        AnsiString mCommand;
        char tmppath[100];
        PROCESS_INFORMATION piProcInfo;
        STARTUPINFO siStartInfo;
        siStartInfo.cb=sizeof(STARTUPINFO);
        siStartInfo.lpReserved=NULL;
        siStartInfo.lpDesktop=NULL;
        siStartInfo.lpTitle=NULL;
        siStartInfo.cbReserved2=0;
        siStartInfo.lpReserved2=NULL;
        siStartInfo.dwFlags=STARTF_USESHOWWINDOW;
        siStartInfo.wShowWindow=SW_HIDE;
        if (SaveDialog1->Execute()){
            GetTempPath(100,tmppath);
            mCommand=AnsiString(tmppath)+"\\arj.exe a "+
                SaveDialog1->FileName+" "+DirectoryListBox1->Directory;
```

```
            proc_create_state=
                CreateProcess(NULL,mCommand.c_str(),NULL,NULL,false,NULL,NULL,NULL,
                &siStartInfo,&piProcInfo);
            if (proc_create_state)
                StatusBar1->Panels->Items[0]->Text = "压缩成功...";
        }
    }
```

上述代码所作的主要工作是创建进程完成文件压缩。要使进程能够完全在后台进行而没有任何显示，关键是设置 siStartInfo 变量的 wShowWindow 域：

```
siStartInfo.wShowWindow=SW_HIDE;
```

这个参数用于指定创建新进程的 ShowWindow 函数的参数，置为 SW_HIDE 使窗口不显示。但仅仅这样还是不行的。非常重要的一点是需要设置 siStartInfo 变量的 dwFlags 域：

```
siStartInfo.dwFlags=STARTF_USESHOWWINDOW;
```

因为如果不做这样设置的话，所创建的进程将不使用 ShowWindow 函数显示窗口。

用于进行解压的函数与上面的函数类似，具体程序代码请参见光盘，这里不再赘述。

9.2 Win32 多线程技术

任何一个应用程序（或进程）都有唯一的一个主线程。应用程序还可能启动其他线程，所有的线程共享进程的虚拟地址空间并能访问全局变量和进程的系统资源。每个程序都有一个或多个线程，线程是应用程序中真正执行的部分。

系统的每一个线程都有叫做 CONTEXT 的结构，一个 CONTEXT 结构是一个包含有线程状态信息的数据结构，具体来说，它包含了线程运行时 CPU 的寄存器状态信息。CONTEXT 结构的定义包含在 winnt.h 中，如下：

```
typedef struct _CONTEXT {
    ............
    DWORDLONG IntRa;   // $26: return address register, ra
    DWORDLONG IntT12;  // $27: temporary register, t12
    DWORDLONG IntAt;   // $28: assembler temp register, at
    DWORDLONG IntGp;   // $29: global pointer register, gp
    DWORDLONG IntSp;   // $30: stack pointer register, sp
    DWORDLONG IntZero; // $31: zero register, zero
    DWORDLONG Fpcr;    // floating point control register
    DWORDLONG SoftFpcr; // software extension to FPCR
    DWORDLONG Fir;     // (fault instruction) continuation address
    DWORD Psr;         // processor status
    DWORD ContextFlags;
    DWORD Fill[4];     // padding for 16-byte stack frame alignment
} CONTEXT, *PCONTEXT;
```

在一个应用程序中，它所有线程的运行顺序是随机的，在一段时间里只有一个线程在执行（排除多个处理器），而这段时间就是一个线程真正的执行时间。当线程进入这一段时间时，它的 CONTEXT 就被载入，使上一次的 CONTEXT 所纪录的寄存器状态被载入 CPU 中。当这一段时间即将结束时，它的 CONTEXT 结构就被当前的 CPU 状态刷新。

以上所述是 Windows 系统实现多线程的具体过程。虽然没有任何两个线程能够同时运行，但由于时间片足够短，感觉上所有线程仍是在同时进行的。

9.2.1 C++ Builder 中实现多线程

Windows API 中通过调用 CreateThread 函数创建一个进程。但在 C++ Builder 中，完全可以使用更加简单的方法。

C++ Builder 将 API 编程对象封装成 TThread 抽象类，TThread 类是 TObject 类的直接后继。在应用程序中，需要从 TThread 派生类并重载 TThread 类相应的方法。每个派生类的实例化都将创建一个新线程。

在 C++ Builder 中可以轻松的实现多线程。具体应遵循以下步骤：

(1) 选择【**File/New**】中的 Thread Object，在对话框中输入类名以产生一个新单元，该单元包含从 TThread 派生类的框架，如图 9-2 所示。

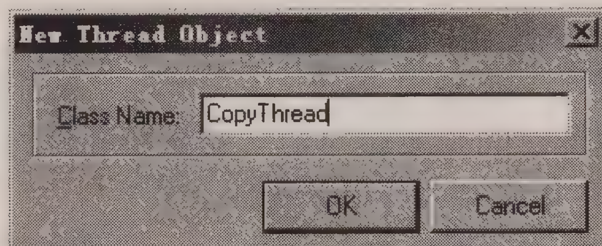


图 9-2　输入 TThread 派生类名

(2) 定义新线程类的构造函数，用以传递必要的参数。例如，一个线程的构造函数重写如下：

```
__fastcall SearchThread::SearchThread(AnsiString mStr[10],AnsiString mPath,bool iSub, int specnum, bool CreateSuspended)
```

构造函数应在头文件和.cpp 文件中都须进行修改。

(3) 在 TThread 派生类的 Execute 方法中加入线程的执行代码。

例如，如果需要在线程中作一些计算，可以如下重载 Execute 方法：

```
void __fastcall TestThrd::Execute()
{
    double mAnswer=0;
    for (long i=0;i<5000000;i++)
        mAnswer+=sin(sqrt(i));
    MessageBox(NULL,("计算结果为："+AnsiString(mAnswer)).c_str(),"计算结果",MB_OK);
}
```

(4) 在主线程中实例化 TThread 派生类，新线程执行。

在本书光盘上提供一个简单的多线程演示程序，该程序演示使用 TThread 类实现多线程编程的方法。图 9-3 给出了简单的多线程演示程序。

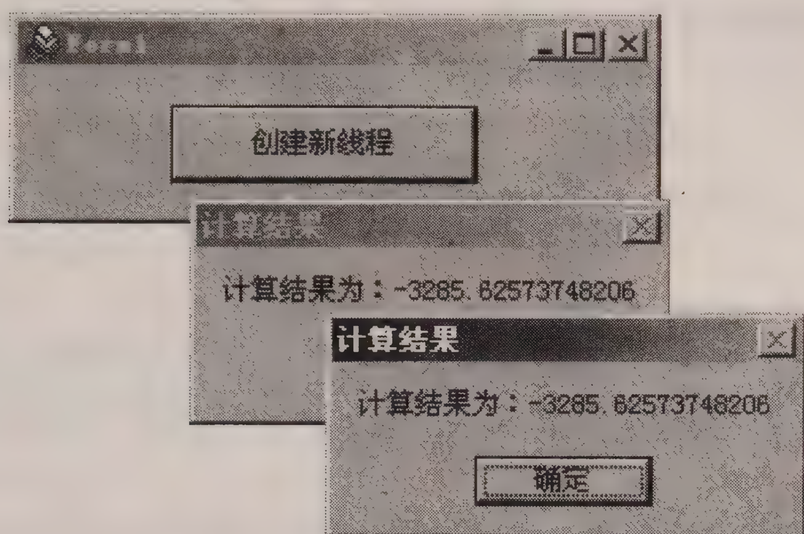


图 9-3　简单的多线程演示程序

9.2.2　TThread 类

以下具体讨论 TThread 抽象类的重要属性和方法。

1. TThread 类的重要属性

- Handle 属性。

线程句柄。利用这个属性，可以通过 WindowsAPI 函数对线程进行控制。

- ThreadID 属性。

线程标识号。

- Priority 属性。

线程优先级。Win32 是抢先多任务的操作系统，系统根据优先级来调度进程。

- Suspended 属性。

为布尔值。若为 true，线程挂起，否则，线程执行。

- FreeOnTerminate 属性。

此属性若为真值，线程结束时线程对象将自动销毁。

2. TThread 类的重要方法

- TThread 方法。

声明：__fastcall TThread(bool CreateSuspended);

TThread 类的构造函数。当 CreateSuspended 的值为 false 时，在类的实例化的同时，会调用 Execute 方法，新线程立即执行。否则新线程挂起，需要用 Resume 方法激活。

- Synchronize 方法。

声明：typedef void __fastcall (__closure *TThreadMethod)(void);

void __fastcall Synchronize(TThreadMethod &Method);

使线程与 VCL 同步的方法，稍后将具体讲述。

C++ Builder 的 TThread 类除了使用简单之外，还有两大优点：其一是它提供一种线程局部存储的替代方法，该方法比原有的方法快 10 倍左右；其二是该类的 Synchronize 方法使得可以在线程中调用 VCL。所以在 C++ Builder 中，利用 TThread 类派生是我们的首选方法。

9.3 实例创建分析

多线程是 Win32 的一项极有特色的技术，多线程技术适合一些可以在后台处理的工作，而前台的用户仍然可以进行通常的操作。另一方面，多线程也可以使应用程序有良好的结构。

在适当的时候使用多线程可以极大的提高应用程序的效率，但它也会带来一些需要解决的新问题。这说明线程并非是万能和十全十美的。在本节中，你将会看到一个多线程应用程序——多线式文件处理工具的实现过程，并处理 C++ Builder 多线程编程的各方面问题。该程序界面如图 9-4 所示。

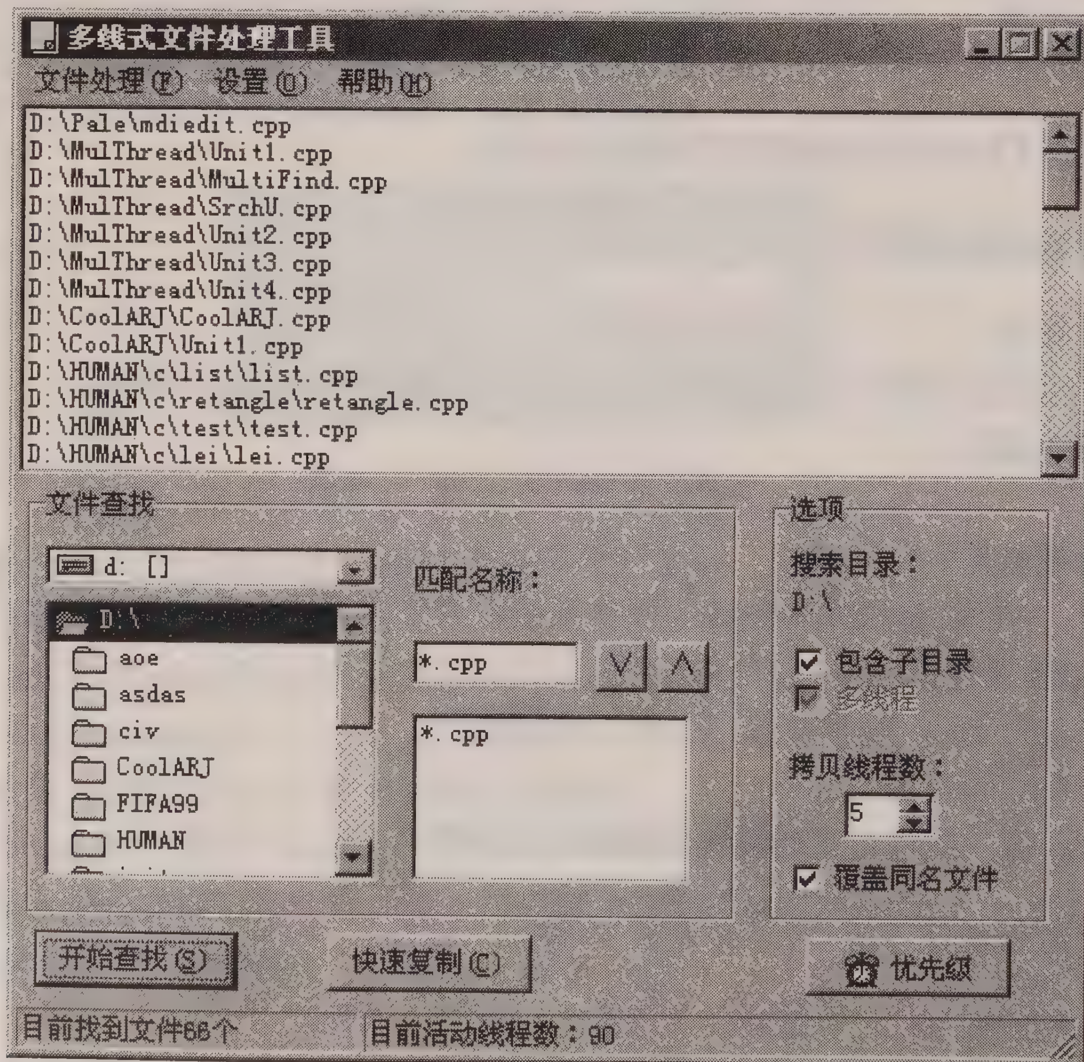


图 9-4　多线式文件处理工具

本章开发的多线式文件处理工具的主要功能是文件查找和文件复制，这两项功能均采用了独特的实现方式——利用多线程技术。

具体程序代码的编制工作大致分为以下几个部分：

- 关于主窗体的一些简单处理。主要是输入查找匹配名称的部分，应编写代码实现匹

配名称往列表框的加入和去除；编写代码使界面上的 CheckBox 和相应的菜单项保持一致。

- 生成 SearchThread 线程框架。修改构造函数，并完成实现查找的代码，关键部分在于 FindFiles 和 SearchDir 两个函数。
- 生成 CopyThread 线程框架。修改构造函数，并完成实现拷贝的代码。
- 再次回到主界面线程，编写代码。完成 SearchThread 线程和 CopyThread 线程的创建和创建前的准备处理工作部分。
- 加入辅助功能。这里包括两块内容：其一，设计"调整优先级窗体"，完成调整优先级需要的相应的代码部分（线程调度的应用）；其二，使程序能够知道线程已全部结束，以提示用户任务完成的功能（线程通信的应用）。

MultiFind 工程主要包含以下一些文件：

- 主窗体单元（主线程）。类名：TForm1，文件：Unit1.h、Unit1.cpp。
- 查找线程单元。类名：SearchThread，文件：Srchu.h、Srchu.cpp。
- 拷贝线程单元。类名：CopyThread，文件：Unit2.h、Unit2.cpp。
- 调整优先级窗体和 About 窗体。

作为一个多线程的例子，多线式文件处理工具有助于更好地理解 Win32 多线程。运行实例，执行查找文件操作，将可以通过状态栏的信息即时看到当前激活的线程数。当查找范围是整个驱动器（如：D:\），并且要查找的又是如"*.cpp"这样可能大量存在的文件时，当前激活线程数的数值会不断变化，而且可能达到 100 以上的峰值。但最终回归为 0（所有线程都执行完毕）。通过查看结果列表框，可以看出多个线程确实是同时执行的，因为不同目录下的文件被交替的显示出来（本例对每个子目录都创建一个查找线程）。

在执行快速拷贝时，由于问题性质不同、设计的算法不同，情况也会有所不同。状态栏显示的激活线程数在一段时间内没有变化，而后一个个的减少，直到为 0（所有线程都执行完毕）。

9.4 实现多线式文件处理工具的技术要点

本节假设读者已经了解了多线程编程的基本方法。因此本节将仅阐述实现多线式文件处理工具的几个要点问题，而对程序细节不作过多解释。

9.4.1 主界面线程

与大多数 Windows 应用程序一样，多线式文件处理工具的主线程是一个与用户交互的界面。如图 9-5 所示。

程序界面主要分为三大部分：上半部分是一个列表框，用于显示找到的文件；文件查找的 GroupBox 中，包括了选择查找目录的部分和输入查找匹配名称的部分，提供选择多个匹配的同时查找；选项的 GroupBox 中，包括了多线文件查找和快速拷贝的控制项。

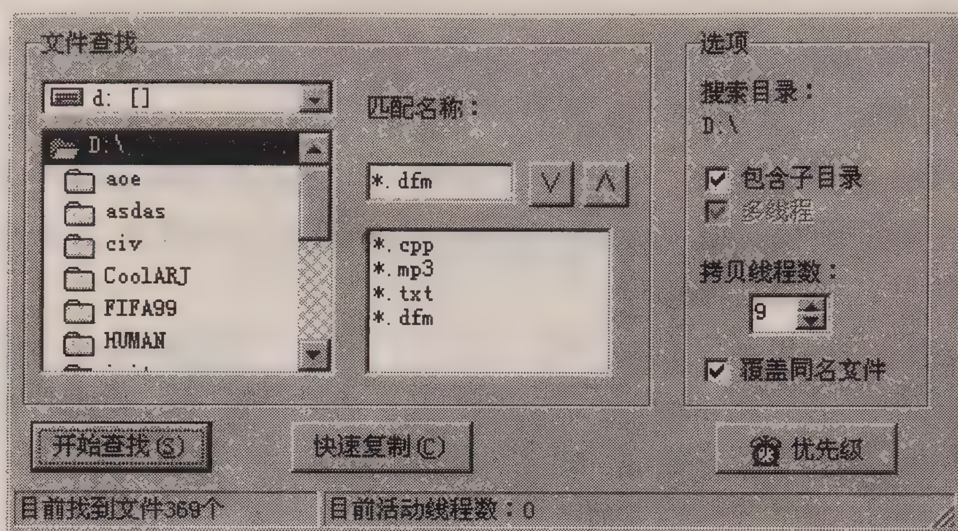


图 9-5　多线式文件处理程序界面的一部分

开始查找和快速复制两个按钮用于启动查找线程或复制线程。开始查找按钮的响应函数如下：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    AnsiString mDir;
    AnsiString mSpec[10];
    int mNum;
    mDir=DirectoryListBox1->Directory;
    if(DirectoryListBox1->
        Directory[DirectoryListBox1->Directory.Length()]!='\\')
    mDir+="\\";
    ListBox1->Clear();
    mNum=ListBox2->Items->Count;
    for (int i =0;i<mNum;i++)
        mSpec[i]=ListBox2->Items->Strings[i];
    ThreadNum=0;
    SearchThread *newsrthrd=new SearchThread(mSpec,mDir,
    CheckBox1->Checked,mNum,false);
    newsrthrd->Priority=mPriority;
}
```

快速复制按钮的响应函数如下：

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    AnsiString dPath;
    int count,num,i=0;
    while(!InputQuery("路径设置", "请输入复制目录:", dPath)||dPath=="");
    if (MessageDlg("全部文件复制到" + dPath+ "?",mtConfirmation,
```

```
                TMsgDlgButtons() << mbYes << mbNo, 0)==mrYes){
                count=ListBox1->Items->Count;
                num=count/(CSpinEdit1->Value-1);
                ThreadNum=0;
                while (i+num<count){
                        CopyThread *newcpthrd =
                                new CopyThread(i,i+num,dPath,CheckBox3->Checked,false);
                        newcpthrd->Priority=mPriority;
                        i+=num;
                }
                CopyThread *newcpthrd =
                        new CopyThread(i,count,dPath,CheckBox3->Checked,false);
                newcpthrd->Priority=mPriority;
        }
}
```

这两个函数的代码很明确，主要获取并处理传递给线程的各个参数，并在处理工作完成后创建线程。

应用程序提供的另一项功能是在列出找到的文件后，在列表框中双击某个文件，该文件将自动打开。具体程序如下：

```
void __fastcall TForm1::ListBox1DblClick(TObject *Sender)
{
        AnsiString FileStr;
        FileStr=ListBox1->Items->Strings[ListBox1->ItemIndex];
        ShellExecute(Handle,"open",FileStr.c_str(),NULL,NULL,SW_SHOWNORMAL);
}
```

实现这一功能所用的函数是 ShellExecute API，它能够自动找到能够打开文件的程序，然后执行这个程序并打开文件。

9.4.2 查找线程

多线式文件处理工具的核心部分之一是文件查找线程。文件查找线程用于搜索一指定目录下符合条件的文件，同时，文件查找线程如果在指定目录下发现了子目录，则将生成一个与本身相同的新线程，用于搜索子目录（如果用户选定了“查找子目录”的话），这是一个递归创建线程的过程。查找线程是一个特殊的线程，线程本身再次创建了新线程，这样的结果将是在同一时间内可能有大量的线程同时被执行。

运行的每一个线程都在后台与应用程序交流信息。作为一个多线程技术的演示，读者将能清晰地看到这些线程的创建和执行过程。

在我们创建的查找线程中，需要定义如下一些私有变量：

```
AnsiString SearchPath;              // 此线程负责搜索的目录
```

```
AnsiString FileSpec[10];        // 目标文件类型
bool FindSub;                   // 是否查找子目录
AnsiString ReportStr;
bool Killit;
int sNum;                       // 查找的文件类型个数
```

还需要更改线程的构造函数，用于传递必要的参数。并在构造函数中，将这些参数赋给相应的变量。查找线程的构造函数如下：

```
__fastcall SearchThread::SearchThread(AnsiString mStr[10],AnsiString mPath,bool iSub, int specnum,
bool CreateSuspended):TThread(CreateSuspended)
{
    SearchPath=mPath;
    sNum=specnum;
    for (int i=0;i<sNum;i++)
        FileSpec[i]=mStr[i];
    FindSub=iSub;
    ReportStr=EmptyStr;
    Killit=false;
}
```

因为创建的程序支持多文件类型同时查找，在查找线程中，FindFiles 和 SearchDir 两个函数是 SearchThread 对象的两个方法。FindFiles 方法用于搜索符合条件的文件，其内容如下：

```
void SearchThread::FindFiles(AnsiString Path)
{
    TSearchRec sr;
    for (int i=0;i<sNum;i++){
        if (FindFirst(Path+FileSpec[i],faArchive,sr)==0){
            do {
                ReportStr=Path+sr.Name;
                Synchronize(Reportit);
                if (Terminated) Killit=true;
            } while(FindNext(sr)==0&&!Killit);
            FindClose(sr);
            Synchronize(SetNumber);
        }
    }
}
```

SearchDir 方法用于查找当前搜索目录中的子目录。每当发现一个子目录，就再建立一个新的 SearchThread 线程搜索，这是一个递归创建线程的过程。函数内容如下：

```
void SearchThread::SearchDir(AnsiString Path)
{
```

```
        TSearchRec sr;
        if (FindFirst(Path+"*.*",faDirectory,sr)==0){
            do{
            if (sr.Attr==faDirectory&&sr.Name[1]!='.'&&!Killit)
                SearchThread *newsrthrd =
                        new SearchThread(FileSpec,Path+sr.Name+"\\",FindSub,sNum,false);
            if (Terminated) Killit=true;
        } while(FindNext(sr)==0&&!Killit);
            FindClose(sr);
        }
    }
```

Execute 方法是每一个线程的入口，该线程中 Execute 方法被重载如下：

```
    void __fastcall SearchThread::Execute()
    {
        FreeOnTerminate=true;
        Form1->ThreadNum++;
        Synchronize(ShowThrdnum);
        if (FindSub)
            SearchDir(SearchPath);
        FindFiles(SearchPath);
        Form1->ThreadNum--;
        Synchronize(ShowThrdnum);
    }
```

注意 ThreadNum 变量在主窗体线程定义，用来纪录当前的线程数。进入线程时执行：

```
    Form1->ThreadNum++;
    Synchronize(ShowThrdnum);
```

计数器加 1，并通过 Synchronize 同步方法调用显示结果的函数。用 Synchronize 实现与 VCL 同步的方法，请看下节的具体阐述。

以上所述是程序中查找线程 SearchThread 的实现要点，从中可以了解 SearchThread 的执行过程，具体来说，每一个要搜索的目录都要创建一个线程处理。

9.4.3 与 VCL 同步

在查找线程中，我们多次用到了 Synchronize 方法。Synchronize 方法是一个非常特殊的方法，它是这样被定义的：

```
void __fastcall Synchronize(TThreadMethod &Method)
```

其中 Method 参数是 TThreadMethod 类型的，它指向一个没有参数没有返回值的函数。

Synchronize 方法的作用是使线程与 VCL 同步，仅仅这样说实在是很抽象。所以首先来看看有关 VCL 和线程的问题。

VCL 相当一部分代码并不是线程安全的。具体来说，大多数由 TComponent 所派生的类与线程是不兼容的，它会导致一些异常。所以在线程中，我们并不能直接使用 VCL 的属性和方法，这就意味着访问或修改程序中用户界面组件的代码不能在线程中执行。

而 TThread 类的 Synchronize 方法正是解决以上问题的，例如可以将可能调用 VCL 的代码写到一个函数中，如在查找线程有这样的函数：

```
void __fastcall SearchThread::ShowThrdnum()
{
    Form1->StatusBar1->Panels->Items[1]->Text=
        "目前活动线程数："+AnsiString(Form1->ThreadNum);
}
```

在线程的某个执行部分，用 Synchronize 方法可以使这些 VCL 的操作得以正确地执行。写法如下：

```
Synchronize(ShowThrdnum);
```

Synchronize 方法实现的原理是：将线程暂时有效的结束，并把它当作主线程的一部分来运行。在这期间，可以使用 VCL。当不再使用 VCL 时，再从同步区里跳出。作为一个规则，在线程中调用 VCL 必须用 Synchronize 方法。

在查找线程中，我们必须时常与界面线程（主线程）进行通信，包括报告找到的符合条件的文件（Reportit 函数），报告找到的文件总数（SetNum 函数），报告当前活动的进程总数（ShowThrdnum 函数）。编写这些函数的目的是使用户能够了解线程的执行状态。具体实现程序请读者参看光盘上的有关代码。

9.4.4 线程的终止

在查找线程中，当线程的 Execute 方法执行完毕后就可以认为线程中止了。此时，C++ Builder 函数 EndThread 被调用，接着 ExitThread API 函数被调用，ExitThread 将释放线程栈，回收 API 线程对象。

当使用完 TThread 对象后，应该注意确保对象被撤销，以确保该对象所占用的所用内存和资源被正确回收。在 C++ Builder 中，最简单的办法是将线程对象的 FreeOnTerminate 属性设为 true，这样线程结束时线程对象将自动被销毁。

TThread 对象在线程结束时触发 OnTerminate 事件，在该事件中处理句柄释放 TThread 对象也是可行的。

在线程处于无限循环或不能响应时，可以使用 Win32 API 的 TerminateThread 函数终止一个正在运行的线程。该函数的定义如下：

```
BOOL TerminateThread(
   HANDLE hThread,// 线程句柄
   DWORD dwExitCode     // 线程退出码
  );
```

TThread 的 Handle 属性提供了线程的句柄，可以用类似于下面的语法来调用该函数：

```
TerminateThread(MyThread->Handle,0);
```

9.4.5 拷贝线程

拷贝线程 CopyThread 是程序中另一部分核心代码。与查找线程不同，执行快速拷贝时，线程的个数由用户指定。可以说拷贝线程较查找线程简单，并更容易理解。其 Execute 方法重载如下：

```
void __fastcall CopyThread::Execute()
{
        Form1->ThreadNum++;
        Synchronize(ShowThrdnum);
        FreeOnTerminate=true;
        for (int i=StartIndex;i<EndIndex;i++){
                FileName=Form1->ListBox1->Items->Strings[i];
                CopyFile(FileName.c_str(),(Path+"\\"+ExtractFileName(FileName)).c_str(),!OverWrite);
        }
        Form1->ThreadNum--;
        Synchronize(ShowThrdnum);
}
```

拷贝文件用了 CopyFile 函数，此函数包含在 shellapi.h 头文件中。

9.5 多线程调度和线程通信

9.5.1 优先级和调度

在前面曾经提到，操作系统负责调度各个线程。对于某一个特定的线程调度时间依赖于该线程被赋予的优先级。某一个特定线程的优先级由创建它的进程的优先级类（Priority Class）和线程本身的优先级（Relative Priority）共同决定。

1. 进程的优先级类

进程的优先级类用于描述系统中运行的某个进程的优先级，可以通过 CreateProcess 函数的 dwCreationFlag 参数指定创建进程的优先级。另一方面，也可以动态地调整进程的优先级。优先级的数字值为 4～24 的范围。表 9-1 列出了各个优先级的对应标志和数字值。

表 9-1 进程优先级标志常量

优先级	标志常量	数字值
Idle	IDLE_PRIORITY_CLASS	4
Normal	NORMAL_PRIORITY_CLASS	7～9
High	HIGH_PRIORITY_CLASS	13
Realtime	REALTIME_PRIORITY_CLASS	24

动态调整进程的优先级要用到 Win32 的 GetPriorityClass 和 SetPriorityClass 函数，这两个函数定义如下：

```
DWORD GetPriorityClass( HANDLE hProcess );
BOOL SetPriorityClass( HANDLE hProcess, DWORD dwPriorityClass );
```

例如，可能有如下代码：

```
SetPriorityClass( GetCurrentProcess(),HIGH_PRIORITY_CLASS);
```

此行代码的作用是把自身程序的优先类设置成 High，其中 GetCurrentProcess 用以返回当前进程的句柄。

2. 相关优先级

特定线程的相关优先级是决定线程总优先级的另一因素。一个线程可能有七种相关优先级，由低到高依次为：Idle、Lowest、Lower、Normal、Higher、Highest、TimeCritical。

在 C++ Builder 中，TThread 类的 Priority 属性可用于控制线程的优先级，该属性是一个 TThreadPriority 枚举类型。它的定义如下：

```
enum TThreadPriority {tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical};
```

与进程的优先级类一样，每一种线程的相关优先级都对应一个数字值，并且在 Win32 API 中为每一级定义了常数标志。表 9-2 列出了各个相关优先级的对应标志和数字值。

表 9-2　线程相关优先级标志常量

相关优先级	标 志 常 量	数字值
Idle	THREAD_PRIORITY_IDLE	–15
Lowest	THREAD_PRIORITY_LOWEST	–2
Lower	THREAD_PRIORITY_BLOW_NORMAL	–1
Normal	THREAD_PRIORITY_NORMAL	0
Higher	THREAD_PRIORITY_ABOVE_NORMAL	1
Highest	THREAD_PRIORITY_HIGHEST	2
TimeCritical	THREAD_PRIORITY_TIME_CRITIC	15

在本章范例程序中，提供给用户动态地调整每个线程的优先级的功能。在程序中采用了整体处理，即在调整了优先级之后，所有的新创建线程都将被赋予同样的用户设定的优先级。这是一个演示性的处理，因为在此程序中，总有很多的线程在执行，如果有大量的线程都被设为了高优先级，高优先级的线程还是抢不到更多的时钟周期。所以在本实例中，这种设定除了让系统更不稳定之外别无它用。

在实例中包含一个优先级调整对话框，如图 9-6 所示。

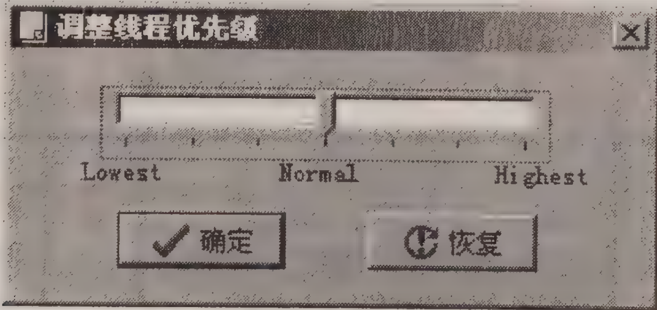


图 9-6　线程优先级调整对话框

而实现线程优先级调整的关键代码如下：

```
void __fastcall TForm4::TrackBar1Change(TObject *Sender)
{
    Form1->mPriority=TThreadPriority(TrackBar1->Position);
}
```

在 Form1 主线程中，每一个生成的线程都被采用 mPriority 的值作为线程的优先级：

```
newsrthrd->Priority=mPriority;
newcpthrd->Priority=mPriority;
```

9.5.2 TEvent 与线程通信

经过上述的工作，多线式文件处理程序已基本完成，它完全达到了最初设定的目标。但是这个工具还有一点小小的缺陷——在查找完成或拷贝完成时，程序无法给出提示。虽然可以通过状态栏所显示的当前活动线程数看出这一点，但程序本身无法知道。

解决这个问题涉及到线程通信的重要概念，这是在多线程编程中常常遇到的问题，两个工作线程或工作线程和主线程之间需要一种手段进行通信。当一个线程完成某种任务时，通知相关线程执行下一个任务。类似于 Visual C++中的 CEvent 类，C++ Builder 中提供了 TEvent 类以解决线程间的通信问题。

TEvent 对象如同一个信号灯，它有两个状态——有信号状态和无信号状态。它一般被创建在应用程序进程空间中，对进程内的全部线程都是可见的。当一个线程发生某种事件后，通过改变事件对象的状态，就可以通知相关线程。也就是说，发送信号的线程将事件对象设置为有信号状态后，等待信号的线程就可以获取这个信号。

在实例多线式文件处理程序中，我们希望在查找或拷贝结束后，主线程能够明确地知道这一点。全部任务结束的判断在线程中是容易的。只需在线程的 OnTerminate 事件的响应函数中判断活动线程数 Form1->TreadNum 是否等于 0。如为 0，则表明当前活动线程只有一个（这是因为在 Execute 函数的结尾 Form1->TreadNum 已被执行，当然线程并未真正完全结束），而且不可能再有新线程被创建（创建线程代码包含在 Execute 函数中），这样就保证了全部线程执行完毕，即任务即将结束。

判断出任务执行完毕后就可以利用 TEvent 类将消息通知主线程，主线程可以做出响应并提示用户。

TEvent 类的使用包括三方面要点。

1. 创建事件对象

一般情况下，事件对象应创建为全局作用对象，以保证程序进程内的所有线程都可以访问此事件对象。TEvent 类的构造函数如下所示：

```
__fastcall TEvent(Windows::PSecurityAttributes EventAttributes, bool ManualReset, bool InitialState, const System::AnsiString Name);
```

图 9-7 显示出了程序的提示功能。

TEvent 类构造函数的第一个参数 EventAttributes 用于设置事件对象的访问权限，一般事

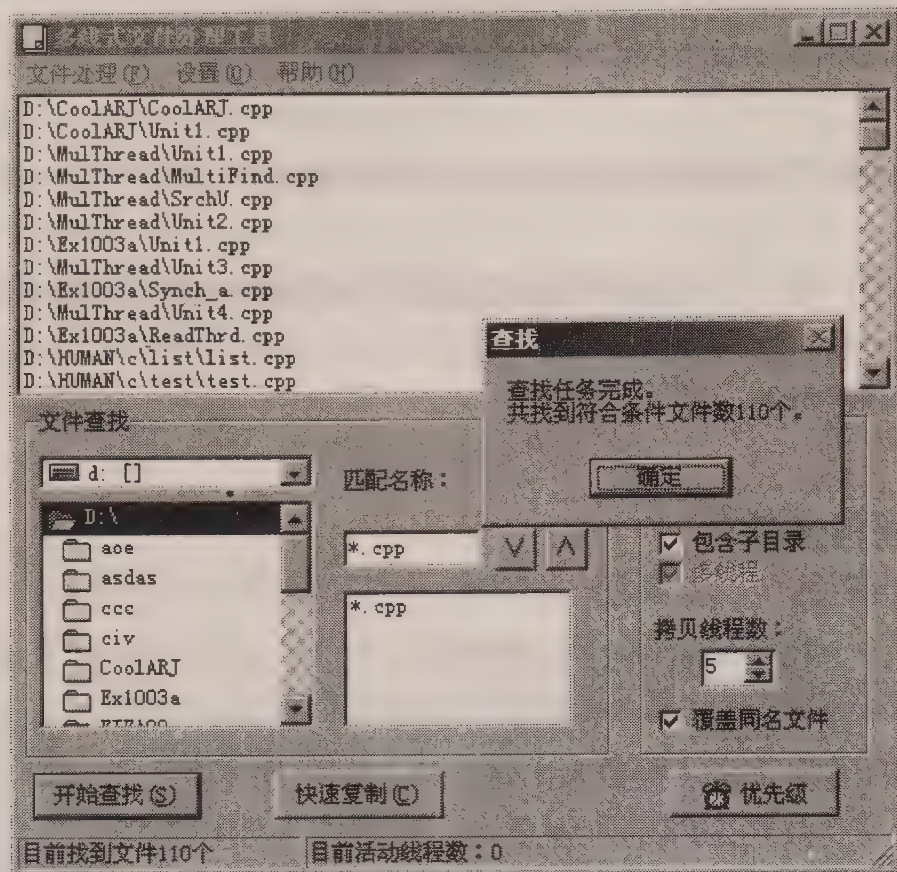


图 9-7 给多线式文件处理工具加入完成任务时做出提示的功能

件可被全部线程访问时，此参数应为 NULL；第二个参数用于指定事件对象的复位方式——手动复位或自动复位，如此参数为 true，则为手动复位——事件信号将一直存在直到有某个线程执行了 ResetEvent 方法，如为 false，则当事件遇到一个正在等待它的线程时，它就自动从有信号状态变为无信号状态；第三个参数 InitialState 用来设置事件对象的初始状态，若此参数为 true，事件对象的初始化状态为有信号状态；第四个参数用于给此事件对象命名。

下面是多线式文件处理程序中两个事件对象的创建：

```
TEvent *SearchFinEvent = new TEvent(NULL,true,false,"SearchEvent");
TEvent *CopyFinEvent = new TEvent(NULL,true,false,"CopyEvent");
```

2. 控制事件对象状态

TEvent 类提供了两种方法用于设置事件对象的状态，即 SetEvent 方法和 ResetEvent 方法，分别用于设置有信号状态和取消有信号状态。

在实例的线程中，可以通过 SetEvent 来标记查找或拷贝完成事件，例如 SearchThread 类的 OnTerminate 响应函数可以这样写：

```
void __fastcall SearchThread::SearchThreadTerminate(TObject  *Sender)
{
    if (Form1->TreadNum==0)
        SearchFinEvent->SetEvent();
}
```

3. 等待事件发生

一个事件发生之后，线程可以通过事件对象通知其他相关线程。那么相关线程如何知道

事件对象的状态发生了变化？相关线程为了获得事件消息，必须使用事件对象的 WaitFor 方法进行等待。

TEvent 对象的 WaitFor 方法声明如下：

TWaitResult __fastcall WaitFor(int Timeout);

Timeout 参数指定等待的毫秒数。此方法返回一个 TWaitResult 类型的值，其取值和相应含义如下：

wrSignaled	在等待期间，信号状态变为有信号状态。
wrTimeout	在等待期间，信号状态无变化。
wrAbandoned	在等待期间，事件对象被删除。
wrError	发生错误。其错误信息可通过事件对象的 LastError 属性得到。

实例程序中，在主线程中作如下处理：

```
while (SearchFinEvent->WaitFor(500)!= wrSignaled);
    MessageBox(NULL,"查找","查找任务完成。",MB_OK);
    SearchFinEvent->Reset();
```

其作用是一直等待，并且每 500 毫秒报告一次，直到查找任务全部完成。

9.6 多线程高级话题

C++ Builder 将 API 多线程编程对象出色地封装成了 VCL 类——TThread 类，这给我们带来了很多方便。不过，尽管 TThread 封装了几乎所有的 API 线程的功能，但还有一些高级功能——特别是对线程之间同步的处理——仍然要使用 API。

TThread 确实是一个 VCL 类，然而在前面提到线程应使用 Synchronize 与 VCL 同步，这好像有些疑惑。其实与 VCL 同步只是一种说法而已。事实上，并非所有的 VCL 操作都必须与主线程同步才能正确执行，TThread 类就是线程级的，其他不从 TCompenent 类派生出来的类也是，对这一类操作我们完全可以不使用 Synchronize 与 VCL 同步就可进行。不过 VCL 中大量的从 TCompenent 类派生出的类是不能在线程中直接操作的。

以下，我们将讨论关于线程的一些高级技术。包括三方面：对线程计时、线程的本地存储和线程间同步。

9.6.1 对线程计时

在多线程环境下，工作线程很难对某个计算所花费的时间计时，因为抢先式多任务操作系统会将 CPU 周期转给其他线程。所以依赖系统时钟并不能对线程中计算所花费的时间进行测量。

为了解决这个问题，Win32 系统提供了 GetThreadTimes 函数，用来获取线程计时的详细信

息。函数声明包含在 WinBase.h 头文件中，如下：

```
BOOL GetThreadTimes(
  HANDLE hThread,          // 线程句柄
  LPFILETIME lpCreationTime,  // 建立时间
  LPFILETIME lpExitTime, // 销毁时间
  LPFILETIME lpKernelTime,          // 系统消耗时间
  LPFILETIME lpUserTime       // 自身代码占用时间
 );
```

其中 hThread 是要获取计时信息的线程句柄。其他参数用于传递索要获得的计时信息，其含义是：

LpCreationTime	线程创建时间。
lpExitTime	线程退出时间，如线程仍在执行，则该值无意义。
lpKernelTime	线程执行操作系统代码花费的时间。
lpUserTime	线程执行自身代码花费的时间。

这四个参数的数据类型是 FILETIME。该结构定义如下：

```
typedef struct _FILETIME {
  DWORD dwLowDateTime;
  DWORD dwHighDateTime;
} FILETIME;
```

这个结构是一个 64 位的值（由 dwLowDateTime 和 dwHighDateTime 组合而成），它代表自 1601 年 1 月 1 日以来的时间，以 100 纳秒为单位。

9.6.2 线程本地存储

当多个线程都要访问全局资源（如全局变量或句柄）时，往往会出一些问题。全局变量在整个程序中是可以共享的，包括在多个线程中。例如，在线程中有一个设置和显示全局变量的函数，可能如下：

```
void SetShowStr(AnsiString Str)
{
    if (Str=="")
        MessageBox(NULL,("The String is "+GlobalString).c_str(),
            "Show GlobalString",MB_OK);
    else
        GlobalString=Str;
}
```

如果设置函数是在一个线程中被调用，它不会有任何问题。可以调用它来设置全局变量的值，而后再次调用来显示全局变量的值。然而，如果同时存在多个线程在任意时刻调用这

个函数设置全局字符串的值，就完全可能出现一个线程调用该函数设置了 GlobalString 的值，而此时被另一个线程抢先调用该函数来设置 GlobalString 的值。此时操作系统可能将 CPU 时间交给第一个线程，该线程的 GlobalString 值就丢失了。这类问题解决的方法是避免在多个线程中处理全局变量，用线程对象的域和参数传递来解决。如果一定要用到全局变量，可以使用 Win32 提供的本地存储的机制。在 C++ Builder 中可以用__thread 关键字来声明全局变量，如：

```
AnsiString __thread GlobalString;
```

这样的声明保证了程序中每一个线程都得到独立的全局变量的拷贝，而使线程中使用全局变量不再发生冲突。

9.6.3 线程同步问题

当使用多线程时，经常需要同步线程对某块数据或资源的访问。最明显的情况是，在进行数据操作时，一个线程是写入操作，另一个线程是读取操作，毫无疑问读操作和写操作是存在确定顺序的。然而，由于两种操作是在不同的线程中进行，系统会认为它们是两个完全无关的任务。如果不加处理的话，其结果可能是读线程读的是不完备或者根本读不出来的数据，而写线程的数据可能写不进去。为了解决这个问题，必须同步两个线程。

Win32 提供多种方法来同步线程。在此将看到如何使用临界区（Critical Sections）、互斥（Mutexes）、信号量（Semaphores）等技术实现线程同步。

为了检验这些技术，来看以下这个例子。这个例子中包括两个数组， Write_Array 和 Read_Array；存在两个线程，ReadThread 和 WriteThread，写线程遍历 Write_Array 数组赋值 1 到 200，读线程将 Write_Array 数组的值依次赋给 Read_Array 数组。线程结束后在列表框里显示最后结果。

这个例子包括三个实现版本：未进行同步的版本、使用临界区的版本和使用互斥的版本。

没有进行同步处理的程序的读线程、写线程和主线程的关键代码如下：

```
// 写线程部分，ReadThrd.cpp
__fastcall WriteThread::WriteThread(bool CreateSuspended)
    : TThread(CreateSuspended)
{
    FreeOnTerminate=true;
}
void __fastcall WriteThread::Execute()
{
    for (int i =0;i<200;i++){
        Form1->Write_Array[i]=i;
        Sleep(2);
    }
```

```
}
// 读线程部分，WriteThrd.cpp
__fastcall ReadThread::ReadThread(bool CreateSuspended)
    : TThread(CreateSuspended)
{
    FreeOnTerminate=true;
}
void __fastcall ReadThread::Execute()
{
    OnTerminate=Form1->ReadFin;
    for (int i =0;i<200;i++){
        Form1->Read_Array[i]=
            Form1->Write_Array[i];
        Sleep(2);
    }
}
// 主窗口线程，Unit1.cpp
void __fastcall TForm1::ReadFin(TObject * Sender)
// 此函数用于响应 ReadThread 类的 OnTerminate 事件。
{
    ListBox1->Items->Clear();
    for (int i=0;i<200;i++){
        ListBox1->Items->Add(
        IntToStr(Read_Array[i]));
    }
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    WriteThread *wt=new WriteThread(false);
    ReadThread *rt=new ReadThread(false);
}
void __fastcall TForm1::FormCreate(TObject *Sender)//初始化数组
{
    for (int i=0;i<200;i++)
        Write_Array[i]=0;
}
```

希望程序依次显示出 0 到 199 的数字，但这就要求读线程在写线程之后被执行。然而由于两线程时同时执行的，数组的内容会被破坏，其结果如图 9-8 所示。

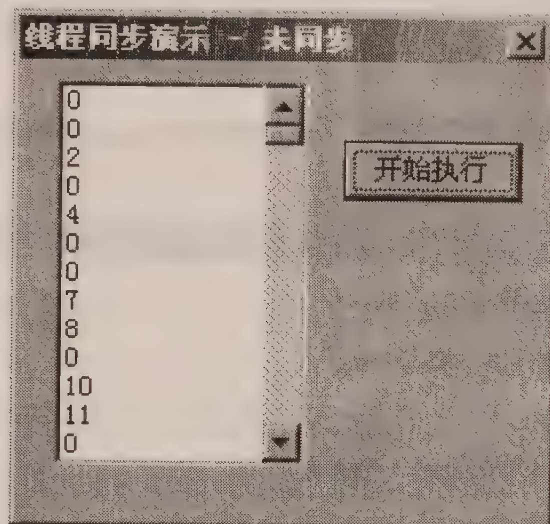


图 9-8　未同步线程运行结果

1. 临界区（Critical Section）

临界区提供了一种最直接的同步线程的方法。一个线程内的临界区会在某一时刻挂起其他线程，而只允许临界区内的代码被执行。将需要同步的代码放到临界区内，就可以实现在一个时段内只有一个线程被执行。

有一系列 API 函数可以支持临界区的设置和操作，它们是：

EnterCriticalSection	进入临界区，实质是指定临界区的起点。
InitializeCriticalSection	初始化临界区，一切关于临界区的操作必需在这之后进行。
LeaveCriticalSection	标出创建临界区的终点。
DeleteCriticalSection	清除临界区对象，释放资源。

可以研究这组 API 函数（另外一个函数是 TryEnterCriticalSection）并用它们来实现临界区。然而对于使用 C++ Builder 编程完全没有必要这么做，因为 C++ Builder 将临界区 API 编程对象封装成了 TCriticalSection 类，使用它可以方便有效地完成临界区的应用。

TCriticalSection 类的属性和方法都比较明了，基本上和 API 函数相同，在此不再赘述。使用 TCriticalSection 类需要在程序中包含 syncobjs.hpp 头文件。

对于上面的程序，可以使用临界区进行同步，修改的代码已经标出。首先应定义 TCriticalSection 对象:

```
TCriticalSection *CS;
```

在写线程中修改代码如下：

```
void __fastcall WriteThread::Execute()
{
    Form1->CS->Enter(); //标志临界区的起点。
    for (int i =0;i<200;i++){
        Form1->Write_Array[i]=i;
        Sleep(2);
    }
    Form1->CS->Leave(); //标志临界区的终点。
```

```
}
```

在主线程中修改代码如下：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    CS=new TCriticalSection(); //初始化，相当于 InitializeCriticalSection
    WriteThread *wt=new WriteThread(false);
    ReadThread *rt=new ReadThread(false);
}
void __fastcall TForm1::ReadFin(TObject * Sender)
{
    ListBox1->Items->Clear();
    for (int i=0;i<200;i++){
        ListBox1->Items->Add(
        IntToStr(Read_Array[i]));
    }
    delete CS; //清除临界区对象，同 DeleteCriticalSection
}
```

当写线程执行到 Form1->CS->Enter()时，其他所有线程被防止进入那块代码，直到执行 Form1->CS->Leave()时，其他线程才有机会被分配到 CPU 周期。运行结果是所希望的正确的输出，如图 9-9 所示。

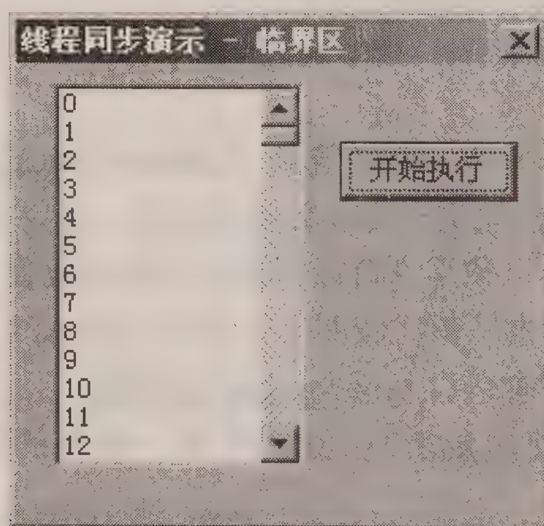


图 9-9　使用临界区完成线程同步

2. 互斥（Mutexes）和信号量（Semephores）

互斥和信号量是另外两种线程同步的方法，它们的功能要比临界区更为强大。但这两种 API 编程对象都没有在 C++ Builder 中封装，所以，只能使用 API 函数来实现这两种技术。

互斥（Mutexes）很像临界区，但有两点区别：其一，互斥可以用于不同进程的线程之间，而临界区只限于本进程的线程之间；其二，创建互斥可以有一个字符串名，引用这个名字可以创建已存在的互斥对象的另一个句柄。

创建互斥对象的函数是 CreateMutex，该函数声明如下:

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
    // 安全属性
  BOOL bInitialOwner,          // 所有标志
  LPCTSTR lpName        // 对象名称
 );
```

lpMutexAttributes 参数表明互斥的安全属性，一般为 NULL，使用缺省的安全属性。

bInitialOwner 指定该线程创建互斥时是否拥有该互斥对象。

lpName 指定互斥对象的名字。如果非空，函数将在系统范围内搜索同名的互斥对象。如果找到，则返回一个已存在的互斥句柄，否则返回一个新的互斥对象的句柄。

使用互斥技术进行同步的关键在于一个名为 WaitForSingleObject 的函数:

```
DWORD WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
);
```

该函数的作用是挂起当前线程 dwMilliseconds 毫秒，直到 hHandle 所指的 API 对象变成 Signaled。是否为“Signaled”对不同的对象有不同的含义，具体到互斥对象来说，Signaled 是指它不被线程所拥有（bInitialOwner 值为 false）。

在本程序中，首先，在创建互斥时 bInitialOwner 置为 false，即互斥不被线程所拥有，处于 Signaled 状态。然后写线程调用 WaitForSingleObject 函数，获取该互斥对象的拥有权，互斥对象状态变成非 Signaled。所以在读线程中，WaitForSingleObject 函数被调用时，读线程将进行等待，直到写线程中已互斥对象句柄作为参数调用 ReleaseMutex 函数时，读线程才能继续被执行。

WaitForSingleObject 函数的 dwMilliseconds 参数可以设为 INFINTE，将一直等待直到对象状态变为 Signaled，这正是我们需要的。

主线程程序代码修改如下：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    hMutex=CreateMutex(NULL,false,NULL);//创建互斥，且为无主
    WriteThread *wt=new WriteThread(false);
    ReadThread *rt=new ReadThread(false);
}
void __fastcall TForm1::ReadFin(TObject * Sender)
{
    ListBox1->Items->Clear();
    for (int i=0;i<200;i++){
        ListBox1->Items->Add(
        IntToStr(Read_Array[i]));
    }
```

```
        CloseHandle(hMutex);//CloseHandle()API 函数用来释放互斥对象
}
```

写线程和读线程代码修改如下：

```
void __fastcall ReadThread::Execute()
{
        OnTerminate=Form1->ReadFin;
        if (WaitForSingleObject(Form1->hMutex,INFINITE)
                ==WAIT_OBJECT_0){
                for (int i =0;i<200;i++){
                        Form1->Read_Array[i]=
                        Form1->Write_Array[i];
                        Sleep(2);
                }
        }
}
void __fastcall WriteThread::Execute()
{
        if (WaitForSingleObject(Form1->hMutex,INFINITE)
                ==WAIT_OBJECT_0){
                for (int i =0;i<200;i++){
                        Form1->Write_Array[i]=i;
                        Sleep(2);
                }
        }
        ReleaseMutex(Form1->hMutex);
}
```

运行结果与利用临界区相同。

WaitForSingleObject 是一个功能强大的函数，它的应用不只在互斥技术中，例如利用信号量实现线程同步也要用到。同时，WaitForSingleObject 函数可以超越进程的界限，甚至可以去等待一个进程。例如下面的程序段，作用是创建一个进程并等待此进程结束。

```
if (CreateProcess(NULL,AppName,NULL,NULL, false,
        CREATE_NEW_CONSOLEl NORMAL_PRIORITY_CLASS,NULL,NULL,
        StartupInfo, ProcessInfo) {
        WaitForSingleObject(ProcessInfo.hProcess,INFINITE);
        GetExitCodeProcess(ProcessInfo.hProcess,Result);
}
```



Win32 API 还提供了 WaitForMultipleObjects 和 MsgWaitForMultipleObjects 函数，应用它们可以做到等待多个对象变成 Signaled 状态。

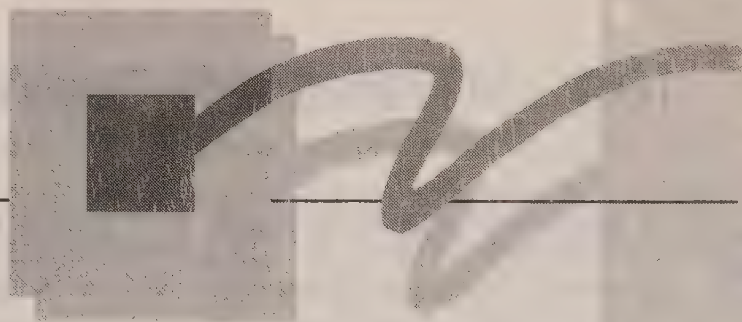
线程同步的另一项技术是使用信号量 API 对象。信号量不仅具有互斥的功能，而且提供了资源计数的能力。也就是说，可以允许在某一时刻有指定数量的线程进入同步代码区，创建信号量的函数是 CreateSemaphores。

应用信号量和应用互斥的方式基本相同，在此不再讲述，有兴趣的读者可以参看相关资料。

第10章

HTML 浏览器
——Internet 相关技术

HTTP 协议和 NMHTTP 组件
创建网页浏览器 HTML 组件
使用 ActiveX 控件
文件传输协议（FTP）编程
邮件协议（SMTP、POP3）编程



本章导读

本章讲述 Internet 协议以及 Internet 客户方应用程序编程技术。所完成的主要实例是一个漂亮且功能齐全的 HTML 浏览器——BCB WebBrowser。这个程序以 Internet Explorer 为蓝本，实现了相同的界面风格、Web 页显示功能以及多种 Web 页浏览和处理功能。我们将采用两种方式实现这个程序的两个不同版本。

此外，本章还将分析邮件协议客户端程序和 FTP 客户端程序的实现。

本章内容包括：

- 使用 HTTP 协议以及 HTML 超文本链接文件的浏览。介绍 C++ Builder 中的 NMHTTP 组件和 HTML 组件。
- 创建 BCB WebBrowser 网页浏览器程序。阐述 HTML 组件的使用。并讨论 TCoolBar 工具栏组件的使用、组件自画以及在 C++ Builder 中使用 ActiveX 控件的问题。
- 使用文件传输协议（FTP）、邮件协议（SMTP、POP3）和特定协议。介绍与之相关的 NMFTP 组件和 NMSMTP 组件等，并创建相关的应用实例。

10.1 HTTP 协议和 CppWebBrowser 组件

Internet 的主要作用是实现不同计算机上信息的互相传输，而各种数据的传输基于不同的协议。HTTP（Hypertext Transport Protocol，超文本传输协议）是 Internet 上最为常用的协议，虽然读者可能对 HTTP 的细节不太了解，事实上我们经常在与 HTTP 打交道，例如各种 Web 浏览器，如 Internet Explorer、Netscape Navigator 等，它们的作用就是通过 HTTP 协议从 Web 服务器中请求 HTML 文档，下载 HTML 文档，然后在客户端解释 HTML 文档并转换为页面显示出来。

在 C++ Builder 中提供了 HTTP 协议相关组件可以方便地实现上述功能。虽然在应用程序中嵌入 Web 浏览器，或在外部启动现成的浏览器程序并没有太大的现实意义，但以下两种应用需求是非常常见的：

- 通过 HTTP 协议检索 HTML 文档，然后按照特定的要求去处理。
- 生成 HTML 格式文件，并在应用程序中显示出来（例如，在应用程序中实现 HTML 格式的帮助系统）。

C++ Builder 提供的 NMHTTP 组件和 CppWebBrowser 组件可以胜任以上两类应用。

10.1.1 使用 CppWebBrowser 组件

HTML（Hyper-Text Markup Language，超文本标志语言）是在 Internet 上应用非常广泛的超文本格式，HTML 是 WEB 浏览器可以读取的标准文档格式。

在版本 5 之前 C++Builder 提供 HTML 组件作为内嵌支持 HTML 语法分析器和页面显示器组件，而 C++Builder 5 以更加强大的 CppWebBrowser 组件替代了原有的 HTML 组件。TCppWebBrowser 组件提供直接进入微软外壳文档对象和控件（Microsoft's Shell Doc Object and Control Library）的功能。实际上，TCppWebBrowser 组件是 Microsoft Internet Control ActiveX 控件（IE 控件）的 VCL 组件形式的封装，该控件包含 IE 4.0 或 IE 5.0 内核，可以像 IE 一样漂亮地显示网页，并支持所有 HTML 扩展。

TCppWebBrowser 事实上封装的是微软外壳文档对象与控件库中的 IWebBrowser2 接口。关于 ActiveX 控件与接口的内容将在本书第 16 章深入讨论。

因为 TCppWebBrowser 是 ActiveX 控件的封装，所以在使用 TCppWebBrowser 组件时必须已注册 Microsoft Internet Control 控件，该控件在安装 InternetExplorer 4 或更高版本时自动注册。

1. CppWebBrowser 组件的主要属性

- Busy 属性。

声明：__property bool Busy = { read = GetWordBoolProp,index=212 };

指出 WebBrowser 控件是否正在进行获取 HTML 文档、解析 HTML 文档或下载 HTML 文档操作。

- Document 属性。

声明：typedef System::DelphiInterface<IDispatch>_di_IDispatch;

__property_di_IDispatch Document={read=GetIDispatchProp,index=203};

重要属性。提供进入当前 HTML 文档的 Document 接口。通过该接口可以进一步分析 HTML 文档，以获取当前 HTML 文档的各种各样的信息。在本书 16 章将会详细讨论该接口。

- FullName 属性。

声明：__property System::WideString FullName={read=GetWideStringProp, index =400};

当前 WebBrowser 控件所在应用程序的完整名称（包含完整路径）。

- LocationName\LocationURL 属性。

LocationName 属性返回当前文档的短名称。例如，如果当前文档是一个 HTML 页面，LocationName 属性将返回 HTML 文档的标题。

LocationURL 属性返回当前文档的 URL 或当前文件全名。

这两个属性均为 WideString 类型。

- Offline 属性。

声明：__property bool Offline = {read=GetWordBoolProp, write=SetWordBoolProp, stored=false, index=550};

离线浏览属性。指定 WebBrowser 是否仅从本地 Cache 中获取 HTML 文档。

- ReadyStates 属性。

声明：__property TOleEnum ReadyState = {read=GetTOleEnumProp, index=-525};

说明 WebBrowser 控件的状态。由于 WebBrowser 组件内部初始化需要一定时间，可以通过该属性获得 WebBrowser 是否确实已准备好。

2. CppWebBrowser 组件的重要方法

- ExecWB 方法。

声明：void __fastcall Exec WB(Shdocvw_tlb::OLECMDID cmdID),

Shdocvw_tlb::OLECMDEXECOPT cmdexecopt, TVariant *pvaIn=TNoParam(), TVariant *pvaOut=TNoParam());

ExecWB 方法用于进行 WebBrowser 组件的某个默认命令，包括打印、保存等等。稍后我们还将深入讨论该方法。

- GoBack\GoForward\GoHome 方法。

WebBrowser 控件分别进行后退、前进、显示主页操作。

- GoSearch 方法。

声明：void__fastcall GoSearch(void);

浏览当前搜索页。当前搜索页的 URL 在系统注册表的 HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main 位置指定。

- Navigate 方法。

声明：void __fastcall Navigate(BSTR URL, Tvariant* Flags=TNoParam(), TVariant* TargetFrameName=TNoParam(), TVariant* PostData=TNoParam(), TVariant* Headers=TNoParam());

CppWebBrowser 组件的重要方法，用于浏览指定 Web 页或指定路径下的文件。其中参数 URL 为指定 Web 页的 URL 或文件路径。

参数 Flags 为浏览标志，可取值为：navOpenInNewWindow、navNoHistory、navNoReadFromCache、navNoWriteToCache、navAllowAutosearch。

参数 TargetFrameName 用于指定 HTML 中需要显示 Frame 名称。

参数 PostData 为使用 HTTP 协议中 POST 协议所包含发送到 Web 服务器的数据。

参数 Headers 包含任何需要发送到 Web 服务器的 header 信息。

- QueryStatusWB 方法。

声明：Shdocvw_tlb::OLECMDF__fastcall QueryStatusWB(Shdocvw_tlb::OLECMDID cmdID);

返回 WebBrowser 控件的某个命令当前是否支持或可用。可配合 ExecWB 方法使用。

- Refresh 方法。

声明：void __fastcall Refresh (void);

重载当前文档。

- Stop 方法。

声明：void__fastcall Stop(void);

停止 WebBrowser 控件当前正在执行的操作。

3. CppWebBrowser 组件的重要事件

- OnDocumentComplete 事件。

当浏览文档显示结束时发生。

- OnDownloadComplete 事件。

当 WebBrowser 控件的某个浏览（Navigate）操作正常结束、中断或失败时发生。

- OnStatusTextChange 事件。

当 WebBrowser 控件的状态条文字改变时发生，通过该事件可以获取控件默认状态条文字信息。

- OnTitleChange 事件。

当 WebBrowser 控件的标题文字改变时发生，通过该事件可以获取控件默认标题文字信息。

10.1.2 使用 NMHTTP 组件

MHTTP 组件与 CppWebBrowser 组件一样可以通过 HTTP 协议下载 HTML 文档，但 NMHTTP 组件并不是一个可视组件，它不能够分析并显示 Web 页面。但 NMHTTP 组件提供对 HTTP 协议完整的支持。它不但可以获取数据，而且能够发送数据；并且支持 Cookie 和代理服务器。应用 NMHTTP 组件实现离线浏览等功能。

1. NMHTTP 组件的重要属性

- Body 属性。

声明：__property AnsiString Body = {read=FBody, write=FBody};

如果 NMHTTP 组件的 InputFileMode 属性为 true，Body 属性返回 HTML 文档的文件名；如果 InputFileMode 属性为 false，Body 属性是下载的 HTML 文档源文件。

- BytesRecvd 属性。

声明：__property int BytesRecvd = {read=FBytesRecvd, nodefault};

这个属性用于返回已经接收到的数据字节数，与 BytesTotal 属性配合使用可以显示传输进度。

- BytesSent 属性。

声明：__property int BytesSent = {read=FBytesSent, nodefault};

这个属性用于返回已经向 HTTP 服务器发送的数据字节数，与 BytesTotal 属性配合使用可以显示传输进度。

- BytesTotal 属性。

声明：__property int BytesTotal = {read=FBytesTotal, nodefault};

这个属性返回要传输数据的字节总数。

- CookieIn 属性。

声明：__property AnsiString CookieIn = {read=FCookieIn};

此属性用于返回 HTTP 服务器发送过来的 Cookie。

- Header 属性。

声明：__property AnsiString Header={read=FHeader,write=FHeader};

如果 NMHTTP 组件的 InputFileMode 属性为 true，Header 属性返回 HTML 文件名。如果 InputFileMode 属性为 false，Header 属性返回实际 HTML 文件的信息头。

- HeaderInfo 属性。

声明：__property THeaderInfo* HeaderInfo = {read=FHeaderInfo, write=FHeaderInfo};

此属性用于指定发给 HTTP 服务器的信息头，在向 HTTP 服务器请求下载 HTML 文档前应该填写此属性。其中，THeaderInfo 的 Cookie 属性指定发送的 Cookie，THeaderInfo 的 LocalMailAddress 属性指定 E-Mail 地址，THeaderInfo 的 LocalProgram 属性指定客户所使用的浏览器名称，THeaderInfo 的 Password 属性指定口令，THeaderInfo 的 Referer 属性用于指定要下载文档的 URL，THeaderInfo 的 UserID 属性用于给出用户名。

- Host 属性/Port 属性。

声明：__property AnsiString Host ={read=ServerName, write=ServerName};

这两个属性分别用于指定要链接服务器的 IP 地址和端口号，Port 属性默认为 80。

- LastErrorNo 属性。

声明：__property int LastErrorNo = {read=GetLastErrorNo, write=SetLastErrorNo, nodefault};

这个属性返回最近一次发生的错误的编号。

- LocalIP 属性。

声明：__property AnsiString LocalIP = {read=GetLocalIP};

返回以点分法表示的客户计算机的 IP 地址。

- Proxy 属性/ProxyPort 属性。

声明：__property AnsiString Proxy = {read=FProxy, write=FProxy};

这两个属性分别用于指定代理服务器的 IP 地址和端口号。

- ReportLeve 属性。

声明: __property int ReportLevel = {read=FReportLevel, write=FReportLevel, default=1};

此属性描述状态信息的详细程度，可取值如下： Status_None、StatusInfo-rmational、Status_Basic、Status_Routines、Status_Debug、Status_Trace。

- Status 特性。

声明：__property AnsiString Status = {read=_Status};

此属性返回当前的状态信息。

2. NMHTTP 组件的重要方法

- Abort 方法。

声明：virtual void __fastcall Abort(void);

此函数用于取消正在发出的请求，相当于 HTML 组件的 Cancel 方法。

- Copy 方法。

声明：virtual void __fastcall Copy(AnsiString URL1, AnsiString URL2);

这个函数把 URL1 参数指定的 HTML 文档复制到 URL2 参数指定的位置，但 HTTP 服务器可能不支持这项功能。

- Delete 方法。

声明：virtual void __fastcall Delete(AnsiString URL);

调用此函数可以删除由 URL 参数指定的 HTML 文档。

- Get 方法。

声明：virtual void __fastcall Get(AnsiString URL);

此函数用于从 HTTP 服务器下载 URL 参数指定的 HTML 文档。如果 InputFileMode 属性为真，下载的 HTML 源文件接收到的数据保存在由 Body 属性所指定的文件中。如果 InputFileMode 属性为假，下载的 HTML 源文件接收到的数据保存在 Body 属性本身。

- Head 方法。

声明：virtual void __fastcall Head(AnsiString URL);

此函数用于获取由 URL 属性所指定 HTML 文档的信息头。

- Post 方法。

声明：virtual void __fastcall Post(AnsiString URL, AnsiString PostData);

调用这个函数将把 PostData 参数指定的数据传送到 URL 参数指定的位置。如果 OutputFileMode 属性为真，PostData 参数是一个文件名，如果 OutputFileMode 属性为假，PostData 参数是一个字符串。

3. NMHTTP 组件使用示例

以下创建一个简单的 NMHTTP 组件使用示例，这个程序支持从 HTTP 服务器下载 HTML

文档和向 HTTP 服务器传输数据。运行程序如图 10-1 所示。

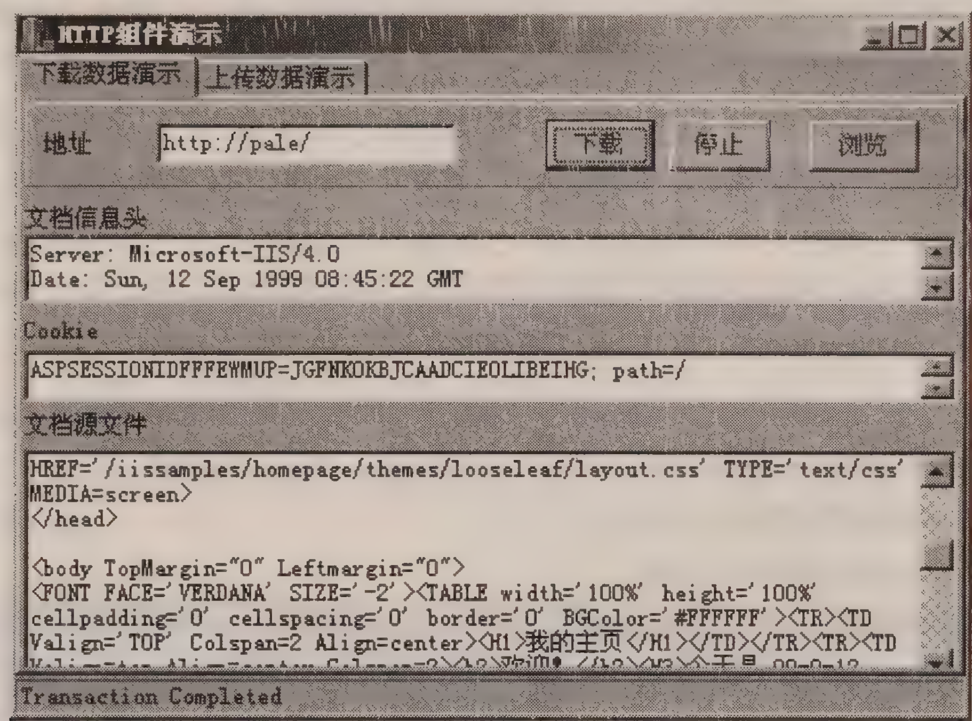


图 10-1　NMHTTP 组件演示程序

在“下载”按钮单击的响应函数包含了下载 HTML 文档的代码。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    NMHTTP1->TimeOut = 5000;
    NMHTTP1->OutputFileMode = false;
    NMHTTP1->InputFileMode = false;
    NMHTTP1->ReportLevel = Status_Basic;
    NMHTTP1->HeaderInfo->UserId = "PALE";
    NMHTTP1->HeaderInfo->Password = "000000";
    NMHTTP1->Get(Edit1->Text);              // 发出获取 HTML 文档的请求
    Memo1->Text = NMHTTP1->Body;       // 显示 HTML 源文件
    Memo2->Text = NMHTTP1->Header;    // 显示 HTML 文档的信息头
    Memo5->Text = NMHTTP1->CookieIn; // 显示 Cookie 内容
}
```

“停止”按钮中断当前的发出请求，响应函数如下：

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
    NMHTTP1->Abort();
}
```

此外，示例程序支持下载的 HTML 文档的显示，“浏览”按钮的响应函数如下。

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    char TmpPath[100];
    GetTempPath(100,TmpPath);
```

```
        // 存储下载的文件
        AnsiString FileName=AnsiString(TmpPath)+"\\Temp.htm";
        Memo1->Lines->SaveToFile(FileName);
        ShellExecute(Handle,"open",FileName.c_str(),NULL,NULL,
            SW_SHOWNORMAL); // 调用默认浏览器显示 HTML 文件
}
```

向 HTTP 服务器传输数据功能的实现代码如下：

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        NMHTTP1->TimeOut = 5000;
        NMHTTP1->OutputFileMode = false;
        NMHTTP1->InputFileMode = false;
        NMHTTP1->ReportLevel = Status_Basic;
        NMHTTP1->HeaderInfo->UserId = "PALE";
        NMHTTP1->HeaderInfo->Password = "000000";
        NMHTTP1->Post(Edit2->Text, Edit3->Text); // 发送指定数据
        Memo3->Text = NMHTTP1->Header;
        Memo4->Text = NMHTTP1->Body;
}
```

另外响应 NMHTTP 组件的 OnStatus 事件可以实现状态信息的显示。

```
void __fastcall TForm1::NMHTTP1Status(TComponent *Sender,
    AnsiString Status)
{
        if (StatusBar1 != 0)
            StatusBar1->SimpleText = Status;
}
```

10.2 创建 BCB WebBrowser 浏览器程序

本节我们将创建一个较为完整的 HTML 浏览器——BCB WebBrowser。这个浏览器应用程序基于 C++ Builder 的 HTML 组件，借助 HTML 组件的强大功能，实现这个范例程序是较为容易的。本节通过此范例程序，演示 HTML 组件各个方面的使用，并讲述在 C++ Builder 中如何实现 IE 风格的应用程序界面。

10.2.1 CoolBar 工具栏

现在看来，Web 浏览器中最为成功的作品是 Microsoft Internet Explorer。IE 的成功之处不仅在于对 HTML 新标准和 HTML 扩展的更好支持，同时它在界面方面也是出色的。Microsoft

从 IE3 开始首次引入了 CoolBar 新风格的工具栏控件，这个控件是 COMCTRLS.DLL 库的一部分。在 C++ Builder 中，提供了 TCoolBar 组件封装了这个崭新的 Windows 通用控件。

TCoolBar 组件与 TToolBar 组件并不类似，TCoolBar 组件不是一个真正的工具栏，而仅是一个工具栏和控件的容器。它本身以子项集合的方式出现，其子项被称为 Band。当选择 CoolBar 组件的 Bands 属性编辑器时，可以建立一个或多个 Band，并设置它们的属性。

每一个 Band 可以放入一个窗口类型的组件，并可以显示一个标题。同时 CoolBar 还支持将背景设为一幅位图，位图将平铺满整个 Band，但这时 CoolBar 上放置的组件应该为透明组件。

放入 CoolBar 的 Band 中的典型组件是工具栏 TToolBar，同时组合框、文本编辑框和动画组件也常常被使用。另外，使用一些小小的技巧，也可以使 Band 中出现一个菜单（像 IE4 中一样），如图 10-2 所示。

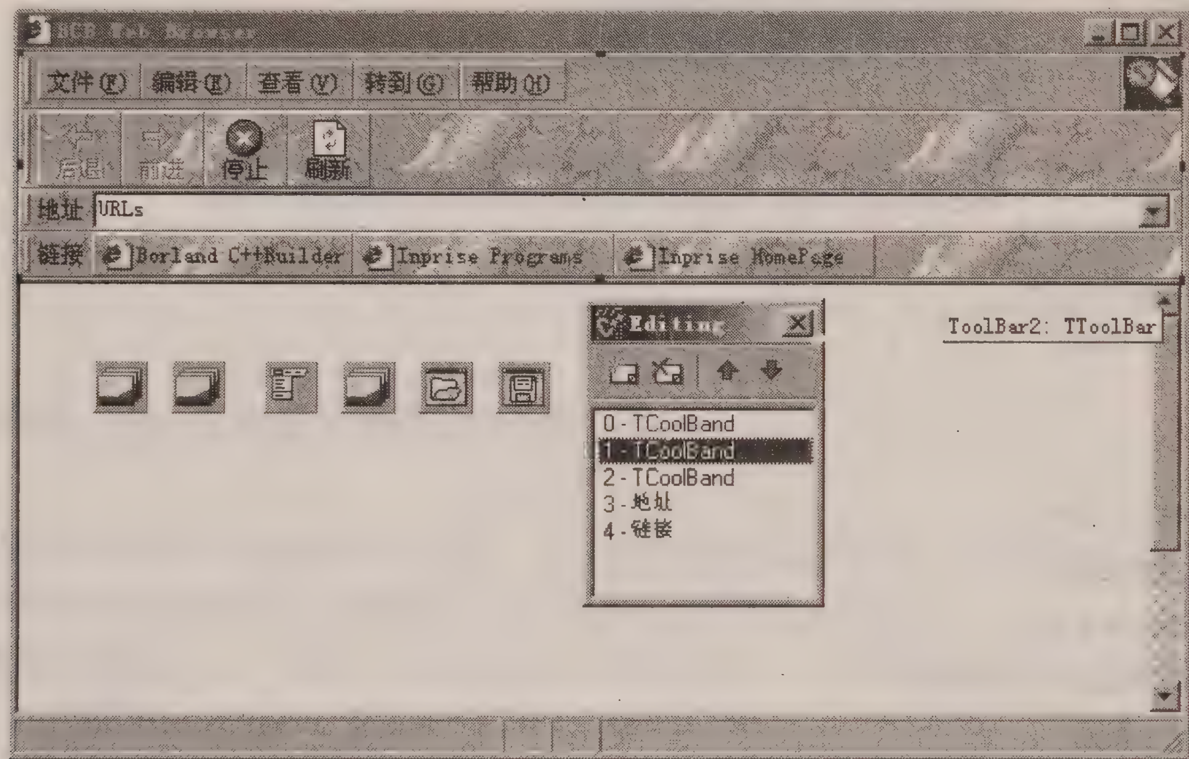


图 10-2 使用 CoolBar 组件的 Bands 属性编辑器

在范例程序 BCB WebBrowser 中共分为 5 个 Band，包括 3 个工具栏，1 个组合框和 1 个动画组件，下面是 BCB WebBrowser 中 CoolBar 组件的属性设置。

```
object CoolBar1: TCoolBar
  Left = 0
  Top = 0
  Width = 604
  Height = 119
  AutoSize = True
  Bands = <
   item
    Break = False
    Control = ToolBar3
    ImageIndex = -1
```

```
      MinHeight = 20
      ParentBitmap = False
      Width = 568
    end
    item
      Break = False
      Control = Animate1
      FixedSize = True
      ImageIndex = -1
      MinHeight = 28
      ParentBitmap = False
      Width = 30
    end
    item
      Control = ToolBar1
      ImageIndex = -1
      MinHeight = 39
      Width = 600
    end
    item
      Control = URLs
      ImageIndex = -1
      MinHeight = 20
      Text = '地址'
      Width = 600
    end
    item
      Control = ToolBar2
      ImageIndex = -1
      MinHeight = 22
      Text = '链接'
      Width = 600
    end>
  FixedOrder = True
  ParentShowHint = False
  Bitmap.Data = {……}
end
```

CoolBar 组件中各个 Band 的布局非常灵活，在设计期间和运行期间都是可调的，但也可以将 FixedOrder 属性设为 true 固定 CoolBar 的布局。

CoolBar 工具栏中的菜单如图 10-3 所示，在 IE4 中，菜单出现在 CoolBar 工具栏中。在

C++ Builder 中，菜单不能放在 CoolBar 的 Band 中，但可以使用 ToolBar 工具栏组件模拟菜单。可以设置工具栏的 ShowCaption 和 Flat 属性为 true，加入与菜单顶级菜单项个数相同的按钮，并将按钮的 AutoSize 属性设为 true，然后创建一个菜单，设置每一个工具栏按钮的 MemuItem 属性为相应的菜单顶级项，其结果如同一个真正的菜单被放入了 CoolBar 工具栏中。

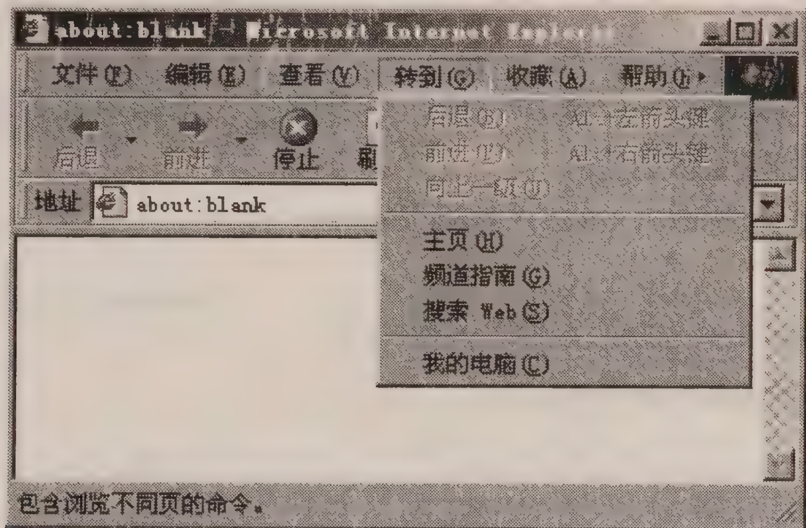


图 10-3　CoolBar 工具栏中的菜单

10.2.2　实现 Web 页的显示和浏览功能

WEB 页浏览和显示功能的实现依靠一个 CppWebBrawser 组件。在创建好 CoolBar 工具栏后，将 CppWebBrawser 组件放入窗体，并设置 Align 属性为 alClient。

要实现用“前进”、“后退”按钮导航，应该定义一个 TStringList 类型的变量纪录历史输入地址，并定义 HistroyIndex 变量纪录当前地址的位置。

```
int HistoryIndex;
TStringList *HistoryList;
```

在窗体的 OnCreate 事件中初始化这两个变量。

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    HistoryList=new TStringList();
    HistoryIndex=-1;
    Animate1->FileName=ExtractFilePath(Application->ExeName)+"bcbwb.avi";
}
```

浏览一个 Web 页的关键是通过调用 CppWebBrowser 组件的 Navigate 方法，当用户在组合框中输入 URL（Uniform Resource Locator）并按 Enter 键后，调用此方法。

```
Void__fastcall TMainForm::RequestDoc(WideString URL)
{
    CppWebBrowser1->Navigate(URL);
}
```

```
void __fastcall TMainForm::URLsKeyDown(TObject *Sender, WORD &Key,TShiftState Shift)
{
    if (Key == VK_RETURN){
        UpdateCombo = true;
        HTML1->RequestDoc(URLs->Text);
    }
}
```

同时，“前进”、“后退”、“刷新”按钮处理函数也将调用此方法。

例如“后退”按钮的响应函数为：

```
void __fastcall TMainForm::BackBtnClick(TObject *Sender)
{
    URLs->Text = HistoryList->Strings[HistoryIndex - 1];
    HTML1->RequestDoc(URLs->Text);
}
```

图 10-4 显示了正在使用 BCB WebBrowser 浏览 HTML 文件。

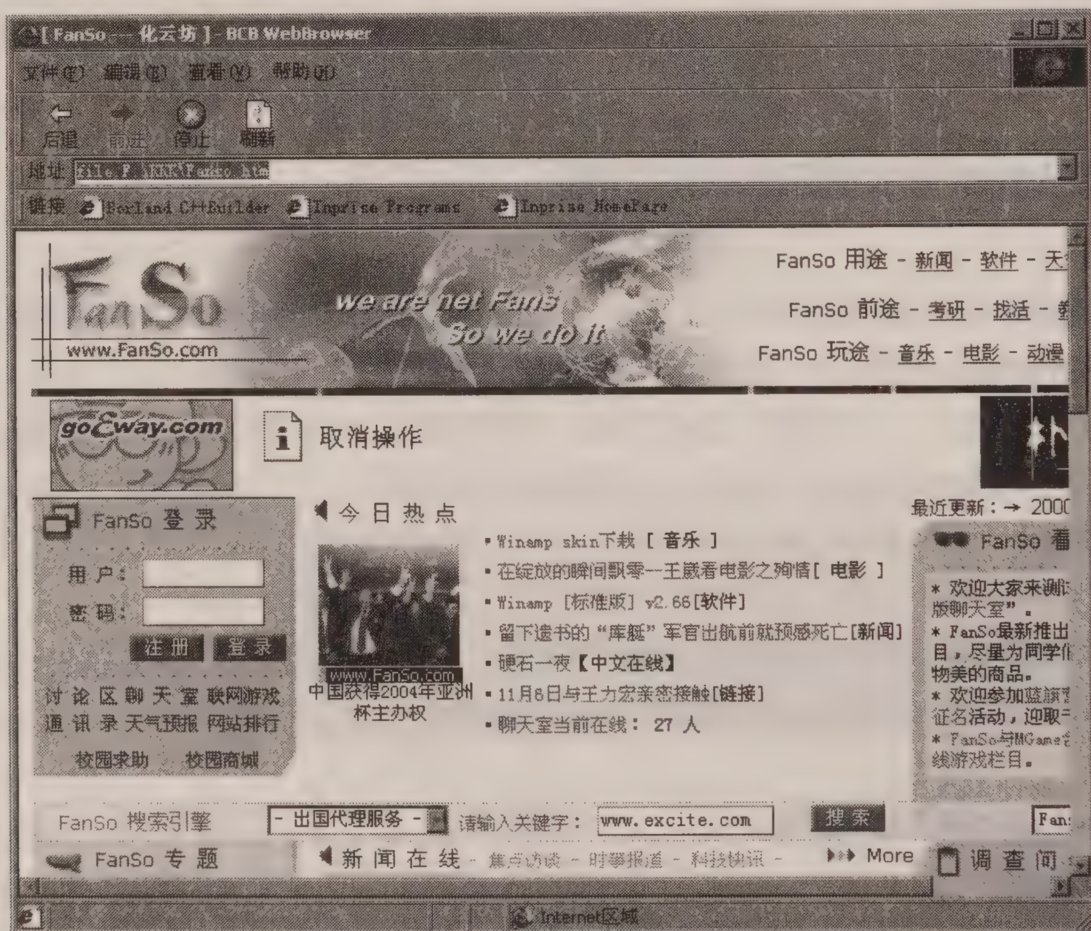


图 10-4 使用 BCB WebBrowser 浏览 HTML 文件

在调用 CppWebBrowser 组件的 Navigate 方法后，处理 CppWebBrowser 组件的 OnNavigateComplete 事件。处理 HistoryList 和 URL 组合框，以及按钮和菜单项。

```
void __fastcall TMainForm::CppWebBrowser1NavigateComplete2(TObject *Sender,
    LPDISPATCH pDisp, Tvariant *URL)
{
    WideString mURL=CppWebBrowser1->LocationURL;
```

```
    {
    int NewIndex = HistoryList->IndexOf(mURL);
    if (NewIndex == -1)
    {
        // 如果是新输入地址
        if (HistoryIndex>=0&&HistoryIndex<HistoryList->Count-1)
            while(HistoryList->Count>HistoryIndex)
                HistoryList->Delete(HistoryIndex);
        HistoryIndex= HistoryList->Add(mURL);
    }
    else
        HistoryIndex = NewIndex;
    if (UpdateCombo)
    {
        // 将地址加入组合框
        UpdateCombo = false;
        NewIndex = URLs->Items->IndexOf(mURL);
        if (NewIndex == -1)
            URLs->Items->Insert(0, mURL);
        else
            URLs->Items->Move(NewIndex, 0);
    }
    URLs->Text = URL;    // 显示 URL
    StatusBar1->Repaint(); // 重画状态栏，在本例中状态栏是自画的
    // 调整按钮和菜单项
    if (HistoryList->Count > 0){
        BackBtn->Enabled = HistoryIndex > 0;
        B1->Enabled = HistoryIndex > 0;
        ForwardBtn->Enabled = HistoryIndex<HistoryList->Count-1;
        F1->Enabled = HistoryIndex<HistoryList->Count-1;
    }
    else {
        BackBtn->Enabled = false;
        B1->Enabled = false;
        ForwardBtn->Enabled = false;
        F1->Enabled = false;
    }
}
```

调用 RequestDoc 方法同时触发 OnBeinRetrieval 和 OnEndRetrieval 事件，程序处理这两个事件调整了 Stop 按钮并打开和关闭动画。

```
void __fastcall TMainForm::HTML1BeginRetrieval(TObject *Sender)
{
    StopBtn->ImageIndex = 4;
    Animate1->Active = true;
}
void __fastcall TMainForm::HTML1EndRetrieval(TObject *Sender)
{
    StopBtn->ImageIndex = 2;
    Animate1->Active = false;
}
```

“停止”按钮、“刷新”按钮的相应函数分别执行 CppWebBrowser 组件的对应方法。

}

最后，响应 HTML 组件的 OnError 事件报告错误。

```
void __fastcall TMainForm::HTML1Error(TObject *Sender, short Number, WideString &Description, int Scode, const WideString Source, const WideString HelpFile, int HelpContext, WordBool &CancelDisplay)
{
    MessageBox(Handle,("BCB WebBrowser 不能正确打开 Internet 站点 "+URLs->Text+"。").c_str(),
        "错误",MB_OK|MB_ICONERROR);
}
```

10.2.3 辅助功能实现

1. HTML 文档的打开与离线浏览

HTML 文档的打开用于浏览本地 HTML 文件，我们需要为本地 HTML 文件构造基于 File 协议的 URL。

菜单项“打开”的响应函数如下：

```
void__fastcall TMainForm::O1Click(TObject *Sender)
{
    if (OpenDialog1->Execute()){
        URLs->Text="file:"+OpenDialog1->FileName;
        UpdateCombo=true;
        RequestDoc(URLs->Text);
    }
}
```

离线浏览功能（脱机工作）的实现可以直接利用 TCppWebBrowser 组件的 Offline 属性，具体代码如下：

```
void__fastcall TMainForm::W1Click(Tobject *Sender)
{
    W1->Checked=!W1->Checked;
    CppWebBrowser1->Offline=W1->Checked;
```

}

2. Web 页保存、打印和剪贴板功能

一些特定的功能，如保存或另存当前 Web 页，打印当前 Web 页，执行复制、剪切、粘贴、全选操作等等，也可以通过 CppWebBrowser 组件所提供的方法轻易实现。这需要利用 CppWebBrowser 组件的 ExecWB 方法。

该方法主要需要传递两个参数 cmdID 和 cmdexecopt。其中 cmdID 可取以下值：

OLECMDID_OPEN	OLECMDID_NEW
OLECMDID_SAVE	OLECMDID_SAVEAS
OLECMDID_SAVECOPYAS	OLECMDID_PRINT
OLECMDID_PRINTPREVIEW	OLECMDID_PAGESETUP
OLECMDID_SPELL	OLECMDID_PROPERTIES
OLECMDID_CUT	OLECMDID_COPY
OLECMDID_PASTE	OLECMDID_PASTESPECIAL
OLECMDID_UNDO	OLECMDID_PEDO
OLECMDID_SELECTALL	OLECMDID_CLEARSELECTION
OLECMDID_ALLOWUILESSSAVEAS	OLECMDID_ZOOM
OLECMDID_GETZOOMRANGE	OLECMDID_UPDATECOMMANDS=21,
OLECMDID_REFRESH	OLECMDID_STOP=23,
OLECMDID_HIDETOOLBARS	OLECMDID_SETPROGRESSMAX=25,
OLECMDID_SETPROGRESSPOS	OLECMDID_SETPROGRESSTEXT=27,
OLECMDID_SETTITLE	OLECMDID_SETDOWNLOADSTATE=29,
OLECMDID_STOPDOWNLOAD	OLECMDID_ONTOOLBARACTIVATED
OLECMDID_FIND	OLECMDID_DELETE
OLECMDID_HTTPEQUIV	OLECMDID_DONTDOWNLOADCSS
OLECMDID_HTTPEQUIV_DONE	OLECMDID_ENABLE_INTERACTION
OLECMDID_ONUNLOAD	OLECMDID_PROPERTYBAG2
OLECMDID_PREREFRESH	OLECMDID_SHOWSCRIPTERROR
OLECMDID_SHOWMESSAGE	OLECMDID_SHOWFIND
OLECMDID_SHOWPAGESETUP	OLECMDID_SHOWPRINT
OLECMDID_CLOSE	

这些命令并不一定被支持或者可能在 WebBrowser 控件的某种状态下不可用。某个命令是否被支持或可用可以通过 CppWebBrowser 组件的 QueryStatusWB 方法来判断。

参数 cmdexecopt 可取以下值：

OLECMDEXECOPT_DODEFAULT	以该命令的默认行为执行
OLECMDEXECOPT_PROMPTUSER	执行命令并要求用户输入信息
OLECMDEXECOPT_DONTPROMPTUSER	执行命令并不进行任何提示
OLECMDEXECOPT_SHOWHELP	显示该命令的帮助

通过参数 cmdexecopt 我们可以决定某些命令的执行方式。

具体到 BCB Web Browser 范例程序中，利用 ExecWB 方法实现多种辅助功能的代码如下：

Source 辅助功能的实现

```
void __fastcall TMainForm::SaveClick(TObject *Sender)
{     //保存 Web 页功能
      CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_SAVE,
           Shdocvw_tlb::OLECMDEXECOPT_DODEFAULT);
}
void_fastcall TMainForm::SaveAsClick(TObject *Sender)
{     //另存为功能
      CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_SAVEAS,
           Shdocvw_tlb::OLECMDEXECOPT_DODEFAULT);
}
void_fastcall TMainForm::PrintClick(TObject *Sender)
{     //打印功能
      CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_PRINT,
           Shdocvw_tlb::OLECMDEXECOPT_DODEFAULT);
}
void__fastcall TMainForm::PageSetupClick(TObject *Sender)
{     //页面设置功能
      CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_PAGESETUP,
           Shdocvw_tlb::OLECMDEXECOPT_DODEFAULT);
}
void__fastcall TMainForm::R1Click(TObject *Sender)
{     //属性功能
      CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_PROPERTIES,
           Shdocvw_tlb::OLECMDEXECOPT_DODEFAULT);
}
void__fastcall TMainForm::CutClick(TObject *Sender)
{     //剪切功能
      CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_CUT,
           Shdocvw_tlb::OLECMDEXECOPT_DODEFAULT);
}
void__fastcall TMainForm::CopyClick(TObject *Sender)
{     //复制功能
      CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_COPY,
           Shdocvw_tlb::OLECMDEXECOPT_DODEFAULT);
```

```
}
void__fastcall TMainForm::PasteClick(TObject *Sender)
{       //粘贴功能
        CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_PASTE,
                Shdocvw_tlb::OLECMDEXECOPT_DODEFAULT);
}
void__fastcall TMainForm::SelectAllClick(TObject *Sender)
{       //全选功能
        CppWebBrowser1->ExecWB(Shdocvw_tlb::OLECMDID_SELECTALL,
                Shdocvw_tlb::OLECMDEXECOPT_DODERFAULT);
}
```

3. 自画状态栏

Internet Explorer 中状态栏不仅显示了文字，并且绘制了图像。在 C++ Builder 中也可以容易的实现这种效果。这需要我们在程序中自画状态栏。

首先，在窗体设计期间设置状态栏中需要自画的 StatusPanel 的 Style 属性设置为 psOwnerDraw。然后响应状态栏的 OnDrawPanel 事件，在状态栏上绘制图像。

```
void __fastcall TMainForm::StatusBar1DrawPanel(TStatusBar *StatusBar,
    TStatusPanel *Panel, const TRect &Rect)
{
        switch(Panel->Index){
        case 0:
                LinkImageList->Draw(StatusBar->Canvas,Rect.Left,Rect.Top,0,true);
                break;
        case 4:
                LinkImageList->Draw(StatusBar->Canvas,Rect.Left,Rect,Top,1,true);
                StatusBar->Canvas->TextOut(Rect.left + 20, Rect.Top+1,"Internet 区域");
        }
}
```

Panel 参数指明了当前自画的 StatusPanel。例如在本例中的状态条共有 5 个 StatusPanel。判断当前自画的 StatusPanel 并做不同的处理。可以看到，程序中第一、五个 StatusPanel 是程序控制字画的，而其他 StatusPanel 则直接用于显示文字。Rect 参数给出了当前自画的范围。有了这些信息，程序就可以通过状态条的 Canvas 属性进行绘制。

程序中通过响应 CppWebBrowser 组件的 OnStatusTextChange 事件，获得并在第二个 StatusPanel 显示 WebBrowser 控件的默认状态条文字。

10.3 BCB WebBrowser 的第 2 版本

使用 C++ Builder 的 HTML 组件可以实现 HTML 文档的下载、分析和显示，它提供了很多

控制功能。但是 HTML 组件有很多缺陷，它与 IE 或 Netscape 的功能相差太远。如果想在应用程序中嵌入更为强大的 HTML 文档浏览功能，可以使用 ActiveX 控件 Microsoft Internet Control。

Microsoft Internet Control 基本上是 Internet Explorer 的核心。如果计算机上安装了 IE 3.0 以上版本，都会存在这个 ActiveX 控件。其实很多商业应用程序都利用这个控件实现嵌入 HTML 文档浏览功能，例如著名的 WebZip 以及最近的金山游侠 II。在 C++ Builder 中，也可以方便地利用这个 ActiveX 控件。

10.3.1 安装 ActiveX 控件

在 ActiveX 控件被安装到计算机上后，如果要在程序中使用，还需要将它安装到 C++ Builder 中。

安装一个 ActiveX 控件首先应在 C++ Builder 的菜单栏中，选择【**Components\Import ActiveX Library**】命令，打开 Import ActiveX Library 对话框（如图 10-5 所示）。

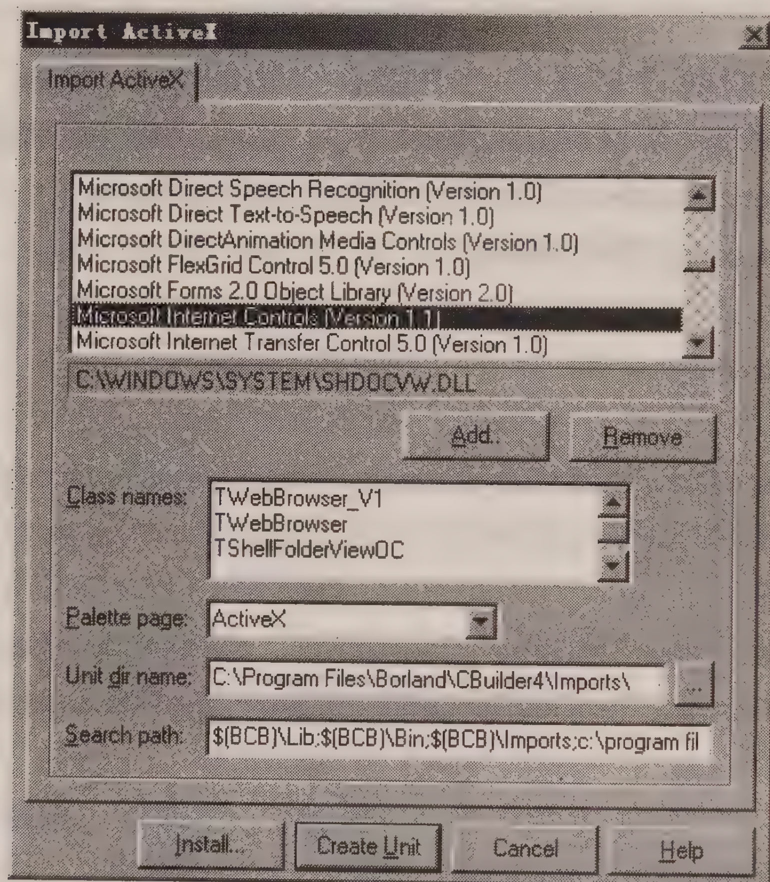


图 10-5 导入 ActiveX 控件对话框

在这个对话框中，可以看到全部在 Windows 中登记的 ActiveX 控件库列表。如果选择一个控件库，C++ Builder 将通过读取该 ActiveX 的类型库列出控件，并为它的单元生成默认文件名，也可以进一步修改此文件名和路径。如果一切正确，选择 Install 按钮，C++ Builder 将为 ActiveX 控件生成 C++单元文件，并将新单元文件添加到 C++ Builder 组件包中，并进一步添加到 C++ Builder IDE 的组件选项板上。

在这里，选择 Microsoft Internet Control 控件库，安装后将可以看到组件选项板的 ActiveX 页（当然，也可以选择安装到其他页）中增加了 WebBrowser 等三个控件。

关于组件包和 ActiveX 控件详细阐述可以参看本书第 13 章和第 16 章。

10.3.2 使用 WebBrowser 控件

C++ Builder 做了一些工作将 ActiveX 控件封装得非常类似于 VCL 组件。可以像使用 C++ Builder 原有组件一样使用 ActiveX 控件，而 ActiveX 控件的属性、方法和事件可以通过 C++ Builder 在安装 ActiveX 控件时生成的 C++源文件看到。

在此以 WebBrowser 控件为基础创建 BCB WebBrowser 的第二版本。由于 Web Browser 控件就是 IE 4.0（或 IE 3.0）的内核，BCB WebBrowser（2nd Edition）程序将可以像 IE 4.0 一样漂亮的显示网页，并支持所有 HTML 扩展。但是 WebBrowser 控件提供的可以常规方式调用的控制要少的多（更多的强大功能需要通过 COM 接口调用，参看本书第 16 章），因此我们不得不在 BCB WebBrowser 的第二版中放弃诸如查看源文件、打印、剪贴板等等功能。

界面设计方面只需对原有界面稍做修改，将原来的 HTML 组件替换为 WebBrowser 控件，并删去一些菜单项。

WebBrowser 控件中与 HTML 组件的 RequestDoc 方法对应的是 Navigate 对象方法。在程序中添加一个 RequestDoc 函数：

```
void __fastcall TMainForm::RequestDoc(WideString URL)
{
    OleVariant v1,v2,v3,v4;
    WebBrowser1->Navigate(URL,v1,v2,v3,v4);
}
```

四个 Variant 参数是必要的，但一般情况可以忽视这些参数的值。

利用 RequestDoc 函数，在上一版本中的大部分代码可以保持不变，包括“前进”、“后退”和“刷新”按钮。

在原来 OnDoRequestDoc 事件响应函数中代码可以简单的拷贝到 WebBrowser 组件的 OnNavigateComplete2 事件的响应函数中，用于处理历史地址纪录。

对于原来处理 OnBeinRetrieval 和 OnEndRetrieval 事件的代码可以拷贝到 OnDownloadBegin 与 OnDownloadComplete 事件的响应函数中。

另外，我们处理两个有用的事件 OnTitleChange 和 OnStatusTextChange，通过这两个事件可以像 IE 一样根据 HTML 文档的标题改变窗口标题和状态栏。

```
void __fastcall TMainForm::WebBrowser1TitleChange(TObject *Sender,BSTR URL)
{
    Caption=AnsiString(URL)+" - BCB WebBrowser [2nd Edition]";
}
void __fastcall TMainForm::WebBrowser1StatusTextChange(TObject *Sender, BSTR URL)
{
    StatusBar1->Panels->Items[1]->Text=URL;
}
```

使用 Microsoft Internet 控件创建的 Web 浏览器如图 10-6 所示。

除了 HTTP 协议之外，还有一些其他常用的 Internet 标准协议，如文件传输协议（FTP）

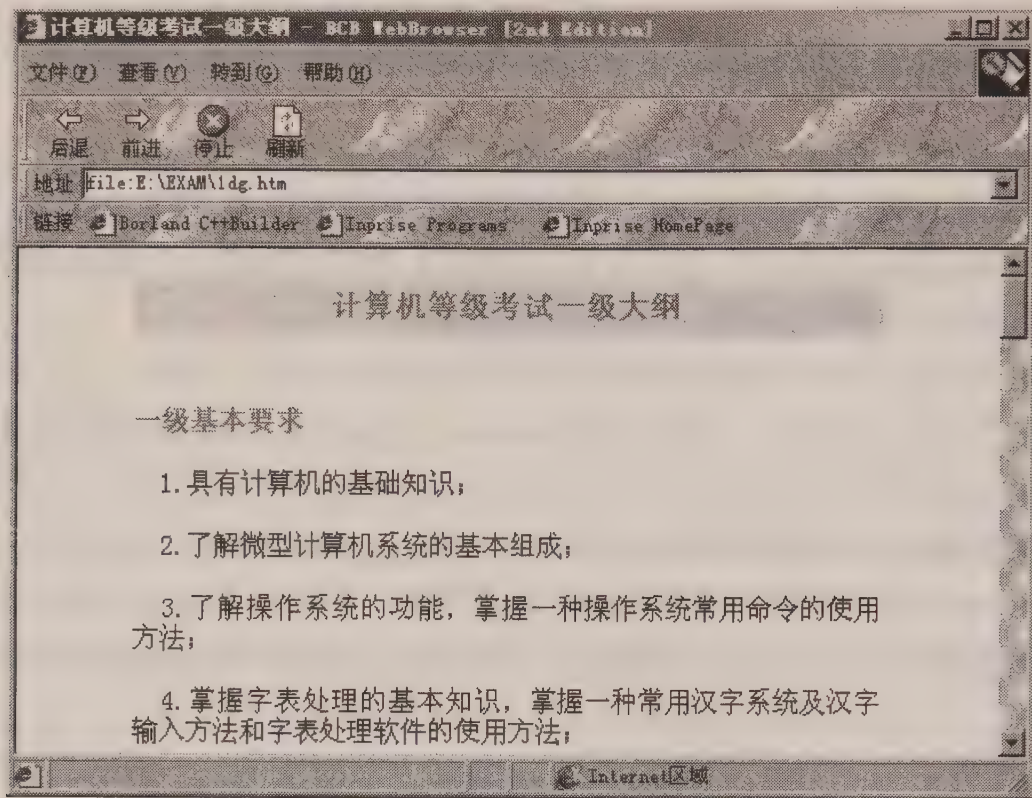


图 10-6 使用 Microsoft Internet 控件创建的 WEB 浏览器

简单邮件传输协议（SMTP）和邮局协议 3（POP3）。现在，已经有相当多的客户端应用程序为文件传输协议和邮件协议提供支持。本节将看到 C++ Builder 中如何实现文件传输协议和邮件协议编程。

10.3.3 使用文件传输协议（FTP）

在 C++ Builder 中包含了 NetMasters 公司建立的 NMFTP 组件，为 FTP 协议编程提供支持。NMFTP 组件功能强大，它对代理服务器和断点续传提供直接支持。NMFTP 组件与 NMHTTP 组件同样是继承于 PowerSock 类，所以 NMFTP 与 NMHTTP 组件有很多属性和方法相同。以下介绍 NMFTP 的一些特殊属性和方法。

1. NMFTP 的特殊属性和方法

- CurrentDir 属性。

声明：__property AnsiString CurrentDir = {read=GetCurrentDir};

此属性返回 FTP 服务器的当前目录

- Password 属性。

声明: __property AnsiString Password = {read=FPassword, write=FPassword};

登录到 FTP 服务器的通常需要给出用户名和口令(一般可以使用 E-mail 地址)，Password 属性用于给出口令。

- UserID 属性。

声明：__property AnsiString UserID = {read=FUserID, write=FUserID};

此属性用于给出用户名。

- ChangeDir 方法。

声明：void __fastcall ChangeDir(AnsiString DirName);

调用此函数请求 FTP 服务器改变当前目录为 DirName 参数指定的目录。

- Download 方法。

声明：void __fastcall Download(AnsiString RemoteFile, AnsiString LocalFile);

下载 FTP 当前目录中的以 RemoteFile 参数为文件名的文件，并保存到本地计算机的指定的文件中，文件名和路径由 LocalFile 参数决定。

- DownloadRestore 方法。

声明：void __fastcall DownloadRestore(AnsiString RemoteFile, AnsiString LocalFile);

调用此方法可以自动进行断点续传。这项较为神秘的技术在 C++Builder 中仅用一个函数就可简单实现，因为 TNMFTP 组件隐藏了所有断点续传实现的技术细节。

- Nlist 方法。

声明：void __fastcall Nlist(void);

调用函数用于返回当前目录中的文件名和目录名列表，列表中的每一项将触发 OnListItem 事件。

- Upload 方法。

声明：void __fastcall Upload(AnsiString LocalFile, AnsiString RemoteFile);

上传一个文件到 FTP 服务器当前目录中，并以 RemoteFile 参数为文件名。

- UploadRestore 方法。

声明：void __fastcall UploadRestore(AnsiString LocalFile, AnsiString RemoteFile, int Position);

这个函数用于实现上传文件的断点续传。与下载文件不同，上传文件需要自已指定续传位置。如果是在 100 字节处断线，则续传时应指定 Position 参数为 101。断点位置可以通过 NMFTP 组件的 BytesSent 属性确定。

2. NMFTP 组件应用举例

在此创建一个简单的 FTP 程序范例，用于演示 NMFTP 组件的使用。这个范例程序支持 FTP 协议的文件上传和下载。它包含了下载文件和上传文件两个页面，其界面如图 10-7 所示。

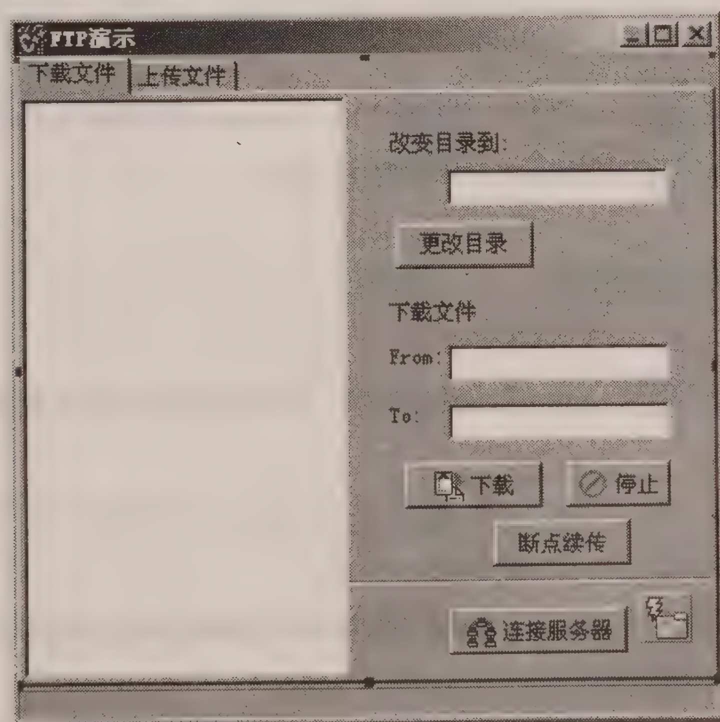


图 10-7　FTP 演示程序的下载文件界面

在进行文件下载和上传之前，首先应该与 FTP 服务器建立连接，为此创建一个连接服务器窗体，如图 10-8 所示。

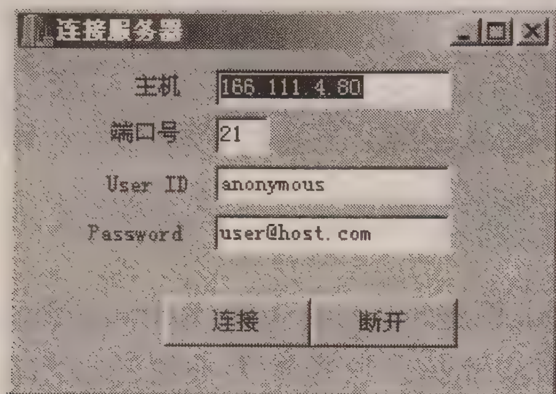


图 10-8　连接服务器窗体

实现连接 FTP 服务器的代码如下：

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
        Form1->NMFTP1->Host = HostTxt->Text;
        Form1->NMFTP1->Port = StrToInt(PortTxt->Text);
        Form1->NMFTP1->UserID = UserTxt->Text;
        Form1->NMFTP1->Password = PassTxt->Text;
        Form1->NMFTP1->Connect();
        Close();
}
```

成功建立连接后，可以进行一系列 FTP 操作。首先，实现更改 FTP 服务器当前目录并列出文件和目录名的功能。代码如下：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        NMFTP1->ChangeDir(Edit1->Text);
        NMFTP1->Nlist();
}
```

Nlist 方法只能触发 OnListItem 事件，处理 OnListItem 事件将具体信息加入 Memo 框。

```
void __fastcall TForm1::NMFTP1ListItem(AnsiString Listing)
{
        Memo1->Lines->Add(Listing);
}
```

实现文件下载只需直接调用 DownLoad 方法，断点续传只需直接调用 DownLoadRestore 方法。

```
NMFTP1->Download(Edit2->Text, Edit3->Text);
NMFTP1->DownloadRestore(Edit2->Text,Edit3->Text);
```

实现上传文件（如图 10-9 所示）应调用 NMFTP 组件的 Upload 方法。

```
NMFTP1->Upload(FileListBox1->FileName, Edit4->Text);
```

另外，程序处理了 NMFTP 组件的 OnPacketRecvd 和 OnPacketSend 两个有用的事件，响应

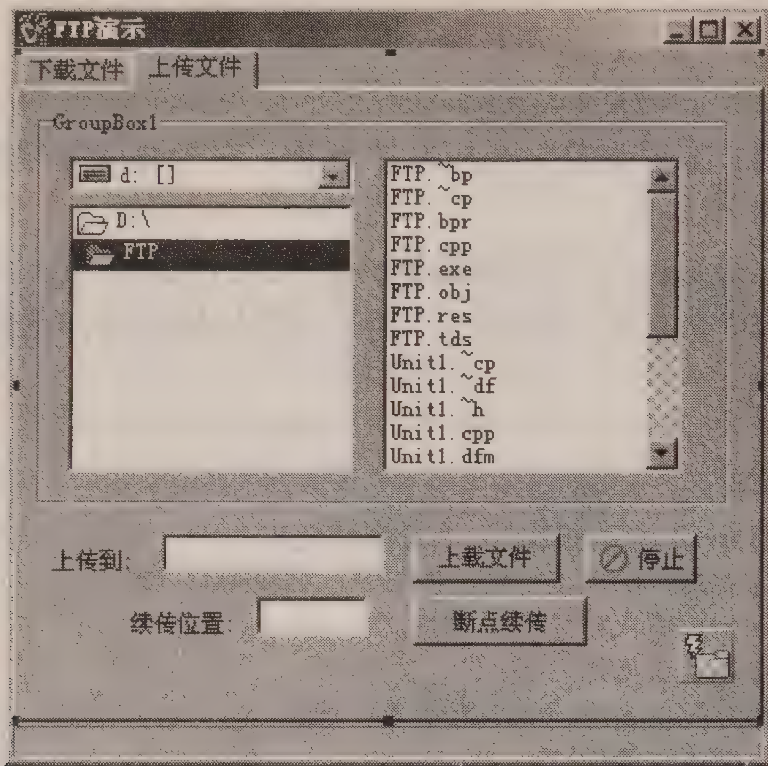


图 10-9　FTP 演示程序的上传文件界面

这两个事件可以实现传输进度的显示。例如 OnPacketRecvd 事件的响应函数如下：

```
void __fastcall TForm1::NMFTP1PacketRecvd(TObject *Sender)
{
    StatusBar1->SimpleText= "目前已下载"+IntToStr(NMFTP1->BytesRecvd)
        +"字节（总共"+IntToStr(NMFTP1->BytesTotal)+"字节）";
}
```

10.3.4　邮件协议和其他特定协议

C++ Builder 提供了一套齐全的组件为 Internet 上的各种协议提供支持。包括 Internet 的 E-Mail 电子邮件协议 SMTP（Simple MailTransfer Protocol，简单邮件传输协议）和 POP3（Post Office Protocol 3，邮局协议 3）。以下将讨论 C++ Builder 中为这两种协议编程提供支持的 NMSMTP 组件和 NMPOP3 组件。

1. NMSMTP 组件

NMSMTP 组件用于实现简单邮件协议（SMTP），该协议用于在 Internet 上发送电子邮件。使用 NMSMTP 组件只需设置主机和端口号（通常 SMTP 协议的端口号为 25），并将发送的内容存储在 PostMessage 属性，然后调用 SendMail 对象方法就可以完成电子邮件的发送工作。

下面具体介绍 NMSMTP 组件的重要属性和方法。

- EncodeType 属性。

声明：__property Nmuue::UUMethods EncodeType = {read=fUUMethod, write=fUUMethod, nodefault};

此属性用于指定邮件附件的编码方式，可取值为 uuMIME 或 uuCode，分别表示按 MIME

编码，或按 UUEncoding 编码。

- Host 属性。

声明：__property AnsiString Host ={read=ServerName, write=ServerName};

此属性用于指定 SMTP 服务器的主机名或 IP 地址。对于发送邮件来说，这是必要的。

- PostMessage 属性。

声明：__property TPostMessage* PostMessage ={read=FPostMessage, write= FPostMessage };

这个属性是 NMSMTP 组件的关键属性，用于纪录传输的邮件信息。为 TpostMessage 类型，此类型包含以下属性：

Attachments	TStringList 类型属性，用于指定附件文件名列表。
Body	TStringList 类型属性，存储邮件正文。
Date	指定邮件的发送日期，如为空，则取当前日期。
FromAddress	指定发信人的 E-Mail 地址。
FromName	指定发信人的名字。
LocalProgram	指定发信人所使用的邮件客户程序。
Subject	存储邮件的标题。
ToAddress	TStringList 类型属性，指定收件人的 E-Mail 地址。
ToBlindCarbonCopy	TStringList 类型属性，指定暗送的 E-Mail 地址。
ToCarbonCopy	TStringList 类型属性，指定抄送的 E-Mail 地址。

- ExtractAddress 方法。

声明：AnsiString __fastcall ExtractAddress(AnsiString TotalAddress);

此函数用于本地检验 E-Mail 地址是否合法。

- SendMail 方法。

声明：void __fastcall SendMail(void);

此函数是 NMSMTP 组件的关键方法，当 PostMessage 属性填写好后，调用此函数将邮件发送出去。

本书配套光盘包含了一个 NMSMTP 组件的示例程序，如图 10-10 所示。

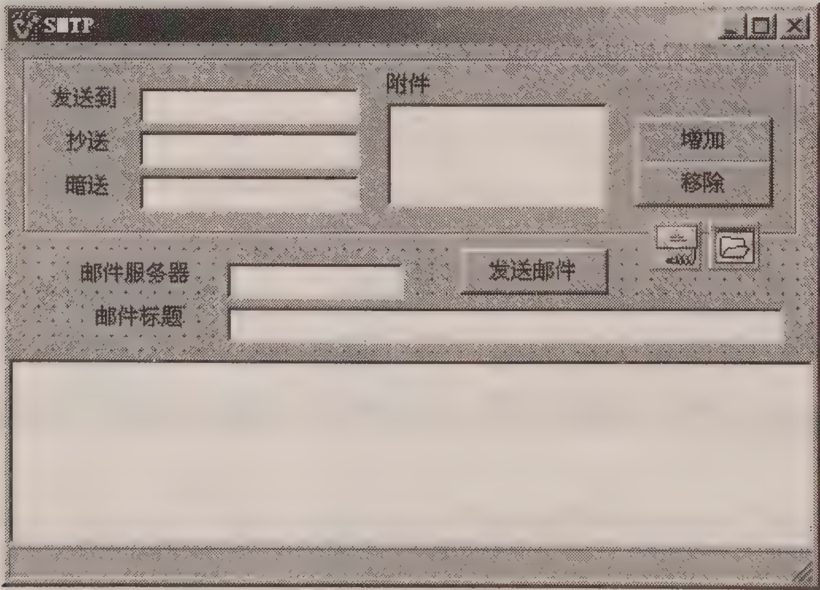


图 10-10　E-Mail 发送演示程序

2. NMPOP3 组件

NMPOP3 组件实现了邮局协议 3（POP3），这个协议用于 Internet 上检索电子邮件，例如查看信箱中有几封信等等。C++ Builder 的 NMPOP3 组件对 POP3 提供了完整的支持，包括支持身份验证登录，信箱邮件信息检索，并支持信箱邮件的删除。

使用 NMPOP3 组件应首先通过 Host 和 Port 属性说明服务器主机，并通过 UserID 和 Password 属性说明用户名和用户口令，之后建立连接。连接成功后，可以通过 MailCount 属性返回信箱中邮件的数目；或通过 MailMessage 和 GetMailMessage 方法实现邮件信息的检索，或通过 Summary 属性和 GetSummary 方法获得邮件的统计信息，以及调用 DeleteMailMessage 函数删除信箱中的一个邮件。

3. 其他特定协议

除了上述几个 Internet 常用协议之外，C++ Builder 的 Internet 组件还包括了其他几种特定协议的组件。它们是 NMDayTime、NMEcho、NMFinger、NMTime 和 NMNNTP 组件。

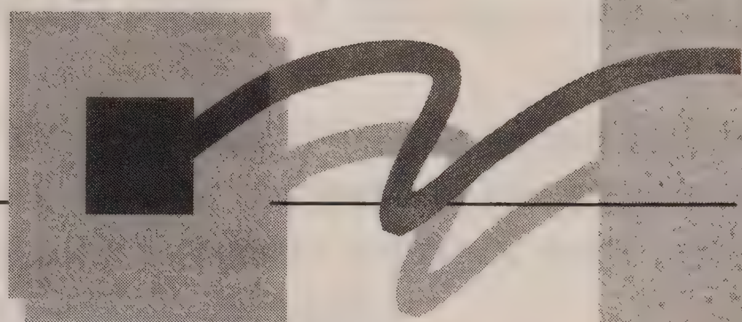
这些协议并不常用，以下对这几个协议作简单介绍：

- NMDayTime 组件和 NMTime 组件可以从服务器上获取日期和时间。获得的日期和时间存储在 DayTimeStr 属性或 TimeStr/TimeInt 属性中。
- NMEcho 组件可以向服务器发送文本信息，并可以从服务器接收应答文本信息。发送文本通过 Echo 对象方法实现。
- NMFinger 组件可以从服务器返回用户的信息。
- NMNNTP 组件用于支持 NNTP（Net News Transfer Protocol）网络新闻传输协议。应用该组件可以登录到 NNTP 新闻服务器，下载可用的新闻组列表，选择新闻组，并提供了新闻阅读和新闻投寄等方法。

第11章

网页留言簿系统
——服务器端 Web 编程

使用 PageProducer 技术创建 Web 页
编写 CGI 程序
利用 WebMoudules 技术创建 ISAPI
输出图形 网站计数器的实现
网络数据库



本章导读

目前已有多种不同Web服务器端程序标准，使用最为广泛的三种标准是CGI、ASP和ISAPI。ASP（Active Server Page）编程简单，但它主要针对脚本语言，并且是解释执行的，因此执行效率有限。本章将讨论使用C++ Builder进行CGI及高效率的ISAPI编程。

本章的目标是创建一个完善的网页留言簿系统。该范例是一个ISAPI应用程序，它通过动态生成HTML页面，支持留言簿数据库的建立、浏览和管理。同时，本章还将讨论并演示建立一些其他常用的服务器应用程序，包括图形化的访问计数器程序、在线考试/问卷系统和基于Web页的聊天室程序。通过这些实例的分析，将使你掌握Web服务器端编程的各种技术。

本章内容包括:

- 使用PageProducer技术定制创建HTML页。
- 编写基本的CGI程序。
- 利用C++ Builder的WebModules技术创建CGI或ISAPI应用。
- 在Web程序中输出图形，创建网页计数器程序。
- 与数据库连接，实现网页留言簿系统。

为了调试和运行本章中提供的基于服务器的例子，你需要一台支持CGI、ISAPI、NSAPI和WIN-CGI（或是它们的结合）的Web服务器。例如Windows NT下的Microsoft Internet Information Server或Windows 95/98下的Personal Web Server都是不错的选择。本章中所有CGI或ISAPI程序都在Personal Web Server 4.0 for Windows 98下运行通过。

11.1 生成 HTML 页面

WWW 的基本结构仍然是客户机/服务器模式，客户端就是 Web 浏览器，如 IE、Netscape 等等，服务器端就是 Web 服务器如 IIS，Web 浏览器和 Web 服务器之间通过 HTTP 超文本传输协议进行通信。HTTP 又是建立在 TCP/IP 协议（Transmission Control Protocol/ Internet Protocol）的基础上。

在 Internet 刚刚出现时，它几乎只含有静态 Web 页。所谓静态 Web 页，是指已经建立好的 HTML 文档，当浏览器选择了一个 URL 时，Web 服务器就把建立好的 HTML 文件传给用户。但更多的时候，动态生成 Web 页面更有价值。动态生成 Web 页面所用的技术就是本章所讨论的 CGI 或 ISAPI 编程技术。但首先将介绍 C++ Builder 中快速建立定制 HTML 页面的 PageProducer 技术。

11.1.1 使用 PageProducer 组件

C++ Builder 4 中包含了一组 HTML 制作器组件，使用这些组件可以方便的建立 HTML 页面，特别是将数据库内容直接转换为 HTML 表格并显示数据。这组组件在开发服务器端 Web 程序时显得非常有意义。

PageProducer 页面制作器组件共包括四个组件，它们位于 C++ Builder 组件选项板的 Internet 页，分别是 TPageProducer 组件、TDataSetPageProducer 组件、TQueryTableProducer 组件和 TDataSetTableProducer 组件。

TPageProducer 组件是一个基本的 HTML 制作器组件，其他三个 HTML 制作器组件都可看作是 TPageProducer 组件的扩展。TPageProducer 组件可以生成带有特定标记的 HTML 文件，这就意味着可以使用 HTML 编辑器（如 FrontPage 98）建立一个 HTML 文件，并在其中要替换为动态数据的地方用类似于<#TagName>的标记替换。而在应用程序中使用 PageProducer 组件处理 HTML 文件，当遇到一个特殊标记时，会触发 OnTag 事件，可以在 OnTag 事件的处理函数中将标记替换为需要替换的数据。

TPageProducer 组件有两个重要的属性 HTMLDoc 和 HTMLFile。

- HTMLDoc 属性。

声明：__property Classes::TStrings* HTMLDoc={read = FHTMLDoc, write = SetHTMLDoc};

这个属性指定一段包含特殊标记的 HTML 文本，PageProducer 组件通过处理这段 HTML 文本生成新的 HTML 文本。

- HTMLFile 属性。

声明：__property System::AnsiString HTMLFile = {read = FHTMLFile,write= SetHTMLFile};

通过这个属性可以指定一个 HTML 文件名，PageProducer 组件通过处理这个 HHTML 文件生成新的 HTML 文本。特别应注意的是，如果 HTMLDoc 属性已经设置，HTMLFile 属性将被忽略。

此外 TPageProducer 组件有一个关键对象方法 Content，此方法用于返回已处理的 HTML

文档。

现在创建一个简单范例 DateToHTML，以演示 TPageProducer 组件的使用。DateToHTML 范例程序用于处理含有<#Date><#Time>标记的 HTML 文件，并将<#Date>替换为系统当前日期，将<#Time>替换为系统当前时间。

该程序借助 TPageProducer 组件的 HTMLFile 属性，当用户指定一个待处理 HTML 文件，调用 TPageProducer 的 Content 方法，这时程序会读取 HTML 文件的代码，并进一步分析它。程序将处理完毕的结果显示在 Memo 框中，如图 11-1 所示。

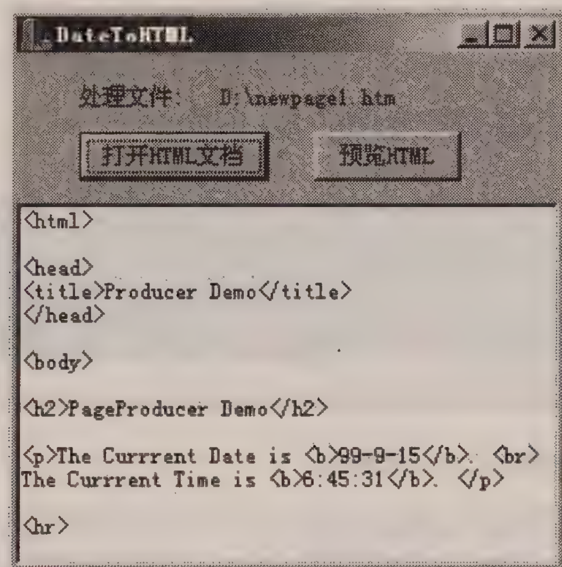


图 11-1　HTML 文件处理器演示程序

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    if (OpenDialog1->Execute()){
        PageProducer1->HTMLFile=OpenDialog1->FileName;
        Label2->Caption=OpenDialog1->FileName;
        Memo1->Clear();
        Memo1->Text=PageProducer1->Content();
    }
}
```

在 TPageProducer 的 OnTag 事件处理函数中，可以通过 TagString 参数获取标记的名称，通过不同标记名称使之替换以不同的 HTML 文本（替换文本赋值给 ReplaceText 引用参数）。

```
void __fastcall TForm1::PageProducer1HTMLTag(TObject *Sender, TTag Tag,const AnsiString TagString,
TStrings *TagParams,AnsiString &ReplaceText)
{
    if (TagString == "Date")
        ReplaceText= DateToStr(Date());
    if (TagString == "Time")
        ReplaceText= TimeToStr(Time());
}
```

运行该程序，打开一个简单的 HTML 文件_newpage1.htm，此文件的内容如下：

```
<html><head>
<title>Producer Demo</title>
</head><body>
<h2>PageProducer Demo</h2>
<p>The Currrent Date is <b><#Date></b>. <br>
The Currrent Time is <b><#Time></b>. </p>
<hr><i><p>This Demo is made by Liubin <#Date>.</i> </p>
</body></html>
```

预览处理后的 HTML 文档，其结果如图 11-2 所示。

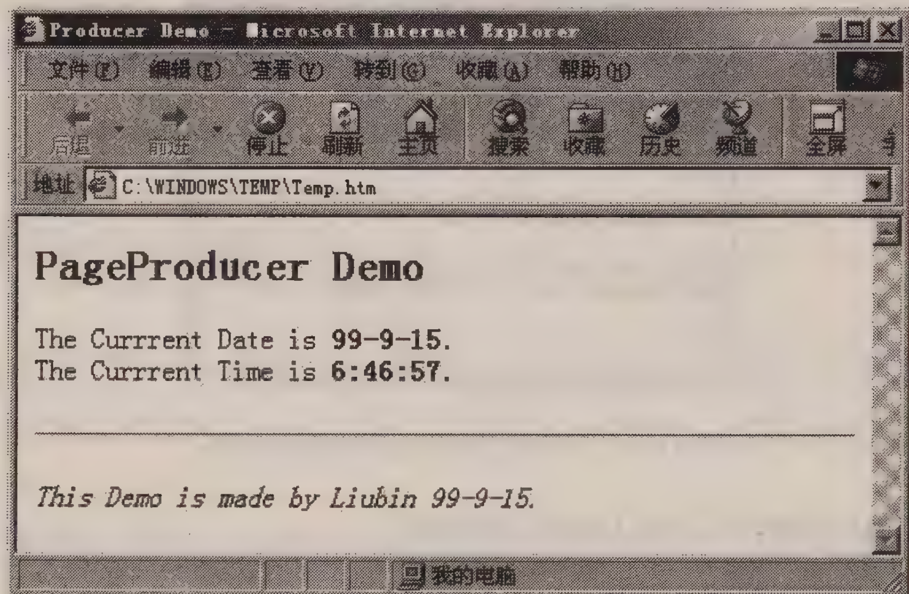


图 11-2　DateToHTML 程序处理后的 HTML 文档

另外，除了 TagString 之外，还可以在标记中加入参数，如类似于<#TagName Param1=Value1 Param2=Value2 ...>的标记。由标记 Param1，Param2 传递的参数值可以通过访问 OnTag 函数的 TagParams 参数得到。

11.1.2　在 Web 页发布数据库

HTML 制作器组件中的另外两个组件——DataSetPageProducer 和 DataSetTableProducer 组件用于将数据库数据以 Web 页形式发布出来。TDataSetPageProducer 可以快速的生成数据页，而 TDataSetTableProducer 用来制作 HTML 格式的数据表格。

TDataSetPageProducer 组件与 TPageProducer 处理 HTML 文档的方式相似，同样是将 HTML 文档中的特定标记替换为实际数据，但 TDataSetPageProducer 组件对数据库提供特定支持，它通过一个 DataSet 属性与一个数据集合相连。这个组件能够自动检测数据表格的字段名称，并可以自动将 HTML 文档中的同名特定标记替换为数据库中的实际数据。

以下是一个使用 TDataSetPageProducer 组件的简单范例。其中 DataSetPageProducer 组件设置如下：

```
object DataSetPageProducer1: TDataSetPageProducer
```

```
    HTMLDoc.Strings = (
     '<HTML><HEAD>'
     '<TITLE>ProducerDemo</TITLE>'
     '</HEAD><BODY>'
     '<H2><CENTER>ProducerDemo：<#NAME></CENTER></H2><BR>'
     '<HR>'
     'Name:<#NAME><BR>'
     'Size:<#SIZE><BR>'
     'Weight:<#Weight><BR>'
     'Area:<#Area><BR>'
     'BMP:<#BMP><BR>'
     '<HR>'
     '<I>This page is made by the program <#program></I>')
    OnHTMLTag = DataSetPageProducer1HTMLTag
    DataSet = Table1
  end
```

Table1 对象连接了一个完整的数据集（在本例中是 DBDDemos 中 Animals.db）。由于应用了 DataSetPageProducer 组件，读者不必手工替换实际数据。以下是 DataSetPageProducer 组件的 OnTag 事件响应函数。

```
void __fastcall TForm1::DataSetPageProducer1HTMLTag(TObject *Sender,
    TTag Tag, const AnsiString TagString, TStrings *TagParams,
    AnsiString &ReplaceText)
{
    if (TagString=="program")
        ReplaceText=ExtractFileName(Forms::Application->ExeName);
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Table1->Next();
    Memo1->Clear();
    Memo1->Text=DataSetPageProducer1->Content();
}
```

使用 TDataSetPageProducer 组件生成的 HTML 文件如图 11-3 所示。

另一个组件 TDataSetTableProducer 可以生成一个 HTML 数据表格来公布一个数据库。这个组件同前两个组件有一点不同，它没有包含 OnTag 事件，DataSetTableProducer 完全是自动的。该组件通过一个 TTable 组件或 TQuery 组件与数据库相连，并自动生成这个标准数据表格页面。

TDataSetTableProducer 组件包含 Columns 属性指定设置数据表格，在设计期间 C++ Builder 提供一个 Columns 属性编辑器用于设置数据表格属性，包括字段属性编辑，标题编辑等等（如图 11-4 所示）。这个编辑器非常易用，并且提供了有效的页面预览功能。

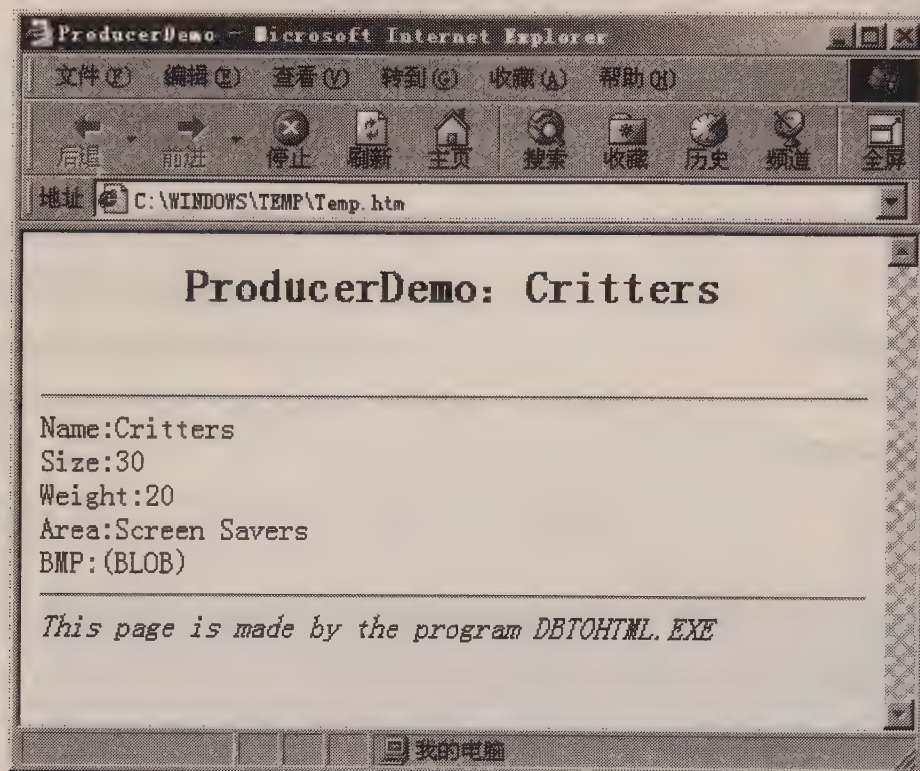


图 11-3　由 DataSetPageProducer 组件生成的数据页

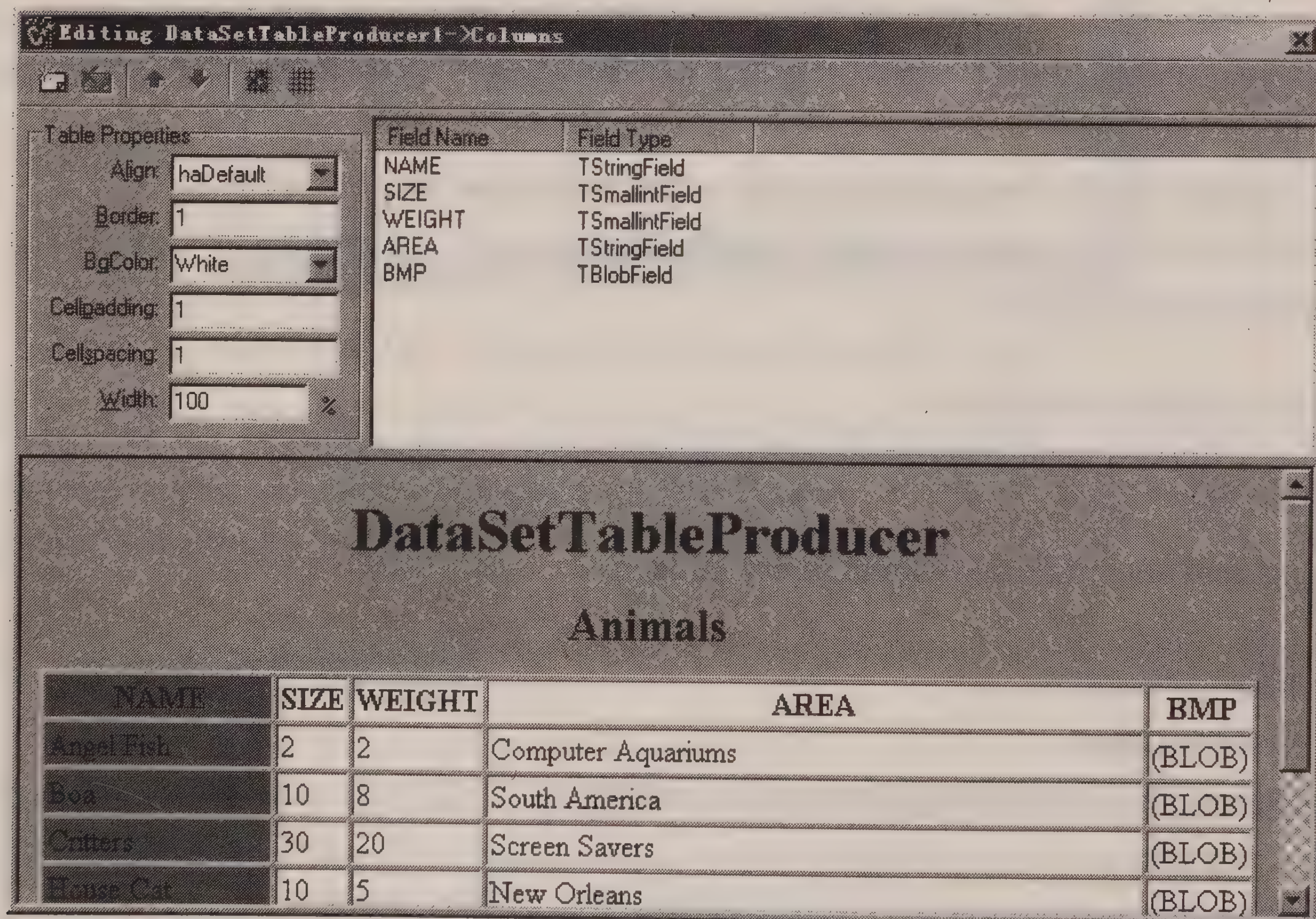


图 11-4　DataSetTableProducer 组件的 Columns 属性编辑器

另外，DataSetTableProducer 组件还允许设定 Header 属性和 Footer 属性，这两个属性分别用于指定位于数据表格之前和之后的 HTML 文档内容。

DataSetTableProducer 组件的 OnFormatCell 事件可以进一步定制表格内容，在这个事件的响应函数中，可以通过 CellRow 和 CellColumn 参数得到当前处理的表格单元位置，并通过 CellData 参数传递处理结果，例如下面一段代码，它处理表格中的每一个 Name 字段内容，为每一个 Name 加入相应链接。

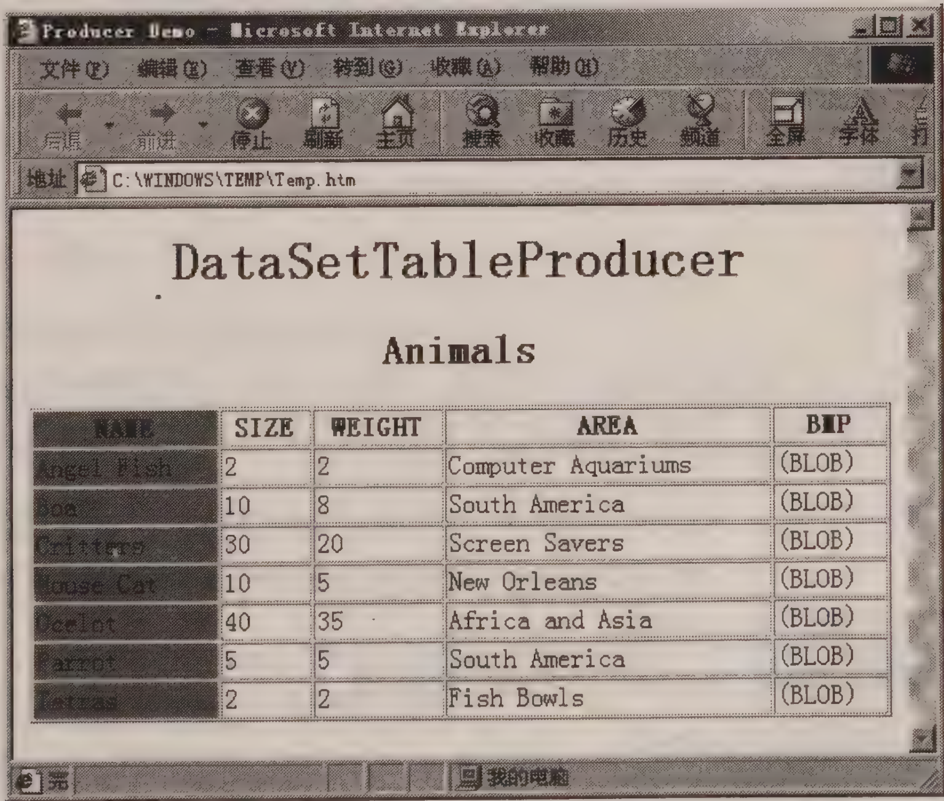
```
void __fastcall TWebModule1::DataSetTableProducer1FormatCell(
    TObject *Sender, int CellRow, int CellColumn, THTMLBgColor &BgColor,
    THTMLAlign &Align, THTMLVAlign &VAlign, AnsiString &CustomAttrs,
    AnsiString &CellData)
{
    if (CellColumn==0&&CellRow!=0){
        CellData="<A HREF=\""+Request->ScriptName+"/show?NAME="+
            Query1->Fields->FieldByName("NAME")->AsString+
            "\">"+CellData+"</A>";
    }
}
```

在 DBToHTML 示例的第二部分演示了 DataSetTableProducer 组件。在程序中，DataSetTableProducer 组件同样与 Table1 相连，按钮“生成表格页面”激活这一应用。

特别应注意的是，DataSetTableProducer 组件并非自动从数据集合的起始位置输出数据表格，而是从数据库的当前位置开始。所以为了输出全部内容，应首先调用 Table 组件对象的 First 方法，如下段代码所示：

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    Memo1->Clear();
    Table1->First();
    Memo1->Text=DataSetTableProducer1->Content();
}
```

可以在图 11-5 中看到程序的输出。当然，可以进一步定制表格，并设法显示出图像，使数据表格更加漂亮。



NAME	SIZE	WEIGHT	AREA	BMP
Angel Fish	2	2	Computer Aquariums	(BLOB)
Boa	10	8	South America	(BLOB)
Critters	30	20	Screen Savers	(BLOB)
House Cat	10	5	New Orleans	(BLOB)
Ocelot	40	35	Africa and Asia	(BLOB)
Parrot	5	5	South America	(BLOB)
Tetras	2	2	Fish Bowls	(BLOB)

图 11-5　基于 DataSetTableProducer 组件的数据表格输出

11.2 创建动态 Web 内容

动态产生 Web 页面工作的一个关键点就是所有处理和操作均在服务器上完成。服务器不再仅仅返回 HTML 文档内容；而是当用户提出一个动态请求时，服务器段程序会根据用户的请求将特定的信息通过生成 HTML 文档发给用户。如果 Web 页信息是不断变化的，动态创建 HTML 页面就显得很有意义。

使用 CGI（Common Gateway Interface，公共网关接口）技术是动态创建 Web 页面的基本并且常用的方法，它也是在这一领域内出现最早的技术。但时至今日，另一些新的 CGI 替代方案纷纷出现，例如 ISAPI（Web 服务器 API）、ASP（Active Server Page）技术，它们有各自的特点并且显得更加有效。

11.2.1 标准 CGI 编程

CGI 是一个用于定义 Web 服务器和外部程序之间的通信方式标准，它使外部程序能够生成 HTML、图像或者其他内容。CGI 是一个定义良好并且被广泛支持的标准，事实上，目前已找不到不支持 CGI 标准的服务器。

从始至终，C/C++语言都是 CGI 编程中一支不可忽视的力量。C/C++的强大的灵活性使它足以胜任这项任务。当 Web 服务器收到一个处理 CGI 的请求时，它以环境变量和标准输入的形式将来自用户的所有信息传给 CGI 应用程序，CGI 程序则通过标准输出发送给用户 HTML。但实际上，在传送 HTML 内容之前，包含生成描述内容是必须的。

下面创建一个最简单的标准 CGI 应用程序。在 Windows 环境中，CGI 应用程序是一个标准控制台程序（Console Application），可以在 C++ Builder 中通过【**File/New**】打开 New Items 对话框并选择 Console Wizard 生成一个控制台应用程序框架并加入如下代码：

```
#include <SysUtils.hpp>
#include <iostream.h>
#include <condefs.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    cout<<"HTTP/1.0 200 OK"<<endl;
    cout<<"CONTENT-TYPE: TEXT/HTML"<<endl<<endl;
    cout<<"<HTML><HEAD>"<<endl;
    cout<<"<TITLE>Hello</TITLE>"<<endl;
    cout<<"</HEAD><BODY>"<<endl;
```

```
        cout<<"<h2>Hello(From CGI Application)</h2>"<<endl;
        cout<<"<hr>"<<endl;
        cout<<"The Current Time is "<<DateTimeToStr(Now()).c_str()<<endl;
        cout<<"</BODY></HTML>"<<endl;
        return 0;
    }
```

这个程序使用 C++的输入输出流进行标准输出。为此，在程序头需要添加#include <iostream.h>一行。该程序中我们的工作仅是在标准输出设备写了几行文本，直接运行这个程序，会在 Dos 窗口看到这些文本。而如果将此程序放在服务器的具有执行属性的目录下（如 cgi-bin，所有的 CGI 或 NSAPI/ISAPI 程序都必须放在具备执行属性目录下才可执行），并在 Web 浏览器中运行这个程序，在浏览器中将出现 HTML 文本，如图 11-6 所示。

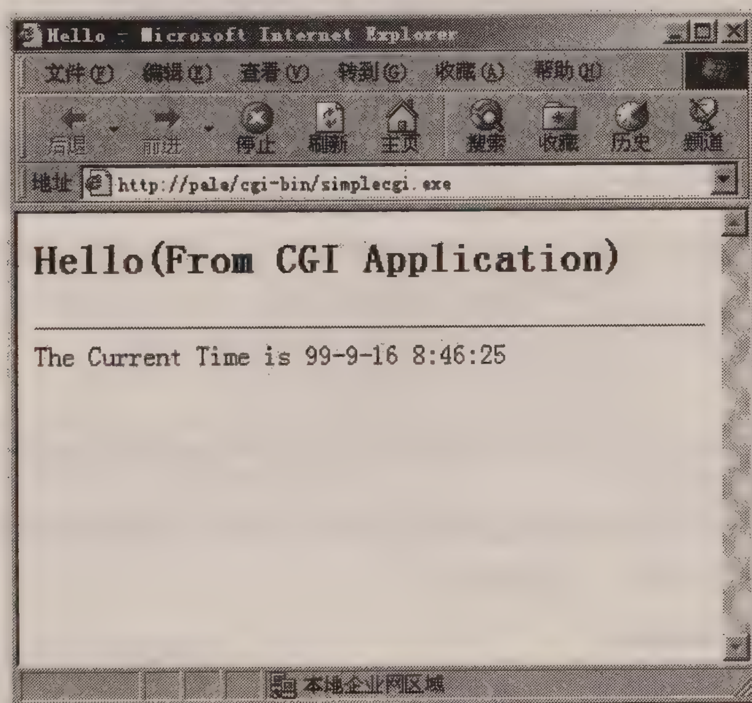


图 11-6 SimpleCGI 演示程序的输出

输出格式中唯一需要特别注意的是，在输出生成描述（CONTENT-TYPE）后必须空一行，才能继续输出 HTML 文档。另外，在 HTML 文件中，可以像连接普通的 HTML 文件那样引用 CGI 应用程序，例如：

```
<a href="\cgi-bin\SimpleCGI.exe">SimpleCGI</a>
```

以上的 CGI 程序确实可以正常运行，但它并不具有与用户交互的功能。实际上，在 HTML 文档中引用 CGI 应用程序时可以传递不同的参数。传递参数的方式如下所示：

```
<a href="\cgi-bin\SimpleCGI.exe?name1=param1">SimpleCGI</a>
<a href="\cgi-bin\SimpleCGI.exe?name1=param2">SimpleCGI</a>
```

在应用程序中，可以通过 argc 参数检测参数的个数，并通过 argv 参数获得参数的具体值。而后可以根据不同的参数输出相应的内容。

CGI 应用程序接受输入的第二种方式是通过环境变量。传递给一个 CGI 应用程序的环境变量很多，表 11-1，11-2，11-3 列举出了一些较为重要并常用的环境变量。

表 11-1　与服务器相关的环境变量

变量	含　义
SERVER_NAME	服务器的 IP 地址或主机名
SERVER_PROTOCOL	服务器用于处理请求的协议名和版本。如 HTTP/1.1
SERVER_PORT	接受 HTTP 请求的端口号，大多服务器为 80
SERVER_SOFTWARE	服务器软件名，例如 WindowsNT

表 11-2　与客户机相关的环境变量

变量	含　义
HTTP_ACCEPT	能被此请求接受的应答模式
HTTP_FROM	客户机的 E-Mail 地址
HTTP_COOKIE	通过远程用户浏览器发送的 Cookies 列表
HTTP_REFERER	链接当前 CGI 程序的文档 URL
HTTP_USER_AGENT	表明客户端软件，并包含版本信息

表 11-3　与请求相关的环境变量

QUERY_STRING	传递给 CGI 程序的 URL 问号后的参数部分
CONTENT_TYPE	被发送数据的 MIME 类型
REQUEST_METHOD	请求所使用的数据传递方法。可以是 Get 或者 Post
PATH_INFO	CGI 程序附加的路径信息
CONTENT_LENGTH	Post 请求向标准输入发送的字节数
REMOTE_ADDR	最终用户的 IP 地址货主机名
SCRIPT_NAME	运行的 CGI 脚本名

在 CGI 应用程序中获得这些环境变量的方法是，使用 GetEnvironmentVariable API 函数，该函数需要包含 winbase.h 头文件，其声明如下：

```
DWORD GetEnvironmentVariable(
  LPCTSTR lpName, // 环境变量名字符串的指针
  LPTSTR lpBuffer,   // 传递环境变量值变量指针
  DWORD nSize        // lpBuffer 指向变量的尺寸
);
```

11.2.2　利用 WebModules 技术创建服务器程序

上一小节我们简单概述了标准的 CGI 编程方法，这是为了使读者对 CGI 编程机制有清楚的了解。然而，使用此种底层的方式编写 CGI 应用程序在任何一个 C/C++开发平台上都可以做到，这显然没有体现出使用 C++ Builder 开发环境的优势。实际上，在 C++ Builder 中提供了 WebModules 服务器应用程序框架，读者马上就会了解到，使用这项技术开发 Web 应用程序是多么方便。

C++ Builder 中的服务器应用程序框架封装了 CGI/Win-CGI/ISAPI/NSAPI 几种服务器端 Web 应用程序的一些底层细节，并使用一种基于 Action 的方便易用的编程模式。对于 CGI 应用程序来说，WebModules 技术的效益可能不太明显，但对于 ISAPI/NSAPI 程序来说，这将节省许多复杂并且相当困难的代码。

在此，我们利用 C++ Builder Web 框架重写上一小节编写的 CGI 程序，并借此演示 C++ Builder 编写 Web 应用程序的方式。

选择【**File/New**】打开 New Items 对话框，在 Object Repository 的 New 页选择 Web Server Application。这将打开 New Web Server Application 对话框，如图 11-7 所示。此对话框提供三个选项——ISAPI/NSAPI、CGI 和 Win-CGI。在此我们选择第二项，C++ Builder 会自动生成 CGI 应用程序框架。

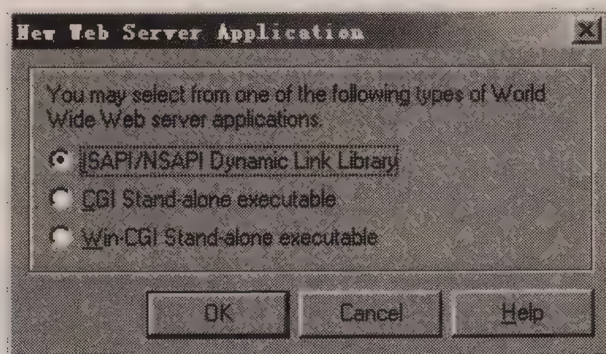


图 11-7　生成一种类型的 Web 应用程序框架

先来分析一下 C++ Builder 生成的服务器应用程序框架。第一，服务器应用程序创建了一个 TWebModule 类的 WebModule 容器，这个容器的地位与普通 Windows 应用程序的主窗体相同，在设计期间，可以将不同的组件（如各种数据库组件，PageProducer 组件等）放入这个容器，并可以在程序中使用它们，第二，服务器应用程序的 Application 对象也不再是 TApplication 类型对象，而是特定的 TCGIApplication 对象，同时入口函数也不再是 WinMain 而是 Main。下面是自动生成的 CGI 应用程序框架的主入口文件 SimpleCGI2.cpp 的一部分：

```
USEFORM("Unit1.cpp", WebModule1); /* TWebModule: DesignClass */
//---------------------------------------------------------------------------
#define Application Httpapp::Application
#pragma link "cgiapp.obj"
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
      try
      {
            Application->Initialize();
            Application->CreateForm(__classid(TWebModule1), &WebModule1);
      Application->Run();
      }
      catch (Exception &exception)
      {
            Sysutils::ShowException(&exception, Sysutils::ExceptAddr());
      }
      return 0;
}
```

TWebModule 对象与 C++ Builder 数据库编程中常用的 TDataModule 对象非常相似，事实上，TWebModule 对象就是从 TDataModule 对象继承下来的。但不同的是，在 C++ Builder 服务器应用程序中，WebModule 对象不仅仅作为一个容器，它还是 Web 服务器应用程序的调度中心。

TWebModule 类继承于 Web 调度器对象 TWebDispatcher 类，并且继承了 TWebDispatcher 类的 Actions 属性。当客户通过 Web 浏览器访问 Web 服务器时，由 Web 调度器对象决定指派哪个动作项（TWebActionItem 对象）去响应客户的请求。

在设计期间可以使用 Actions 属性编辑器，如图 11-8 所示，根据请求的 PathInfo 属性定义一组动作项。这样在应用程序中可以通过编写 OnAction 事件的响应函数，轻松地响应不同路径名的请求。当然，也可以通过忽略 PathInfo 属性定义无参数的动作项。

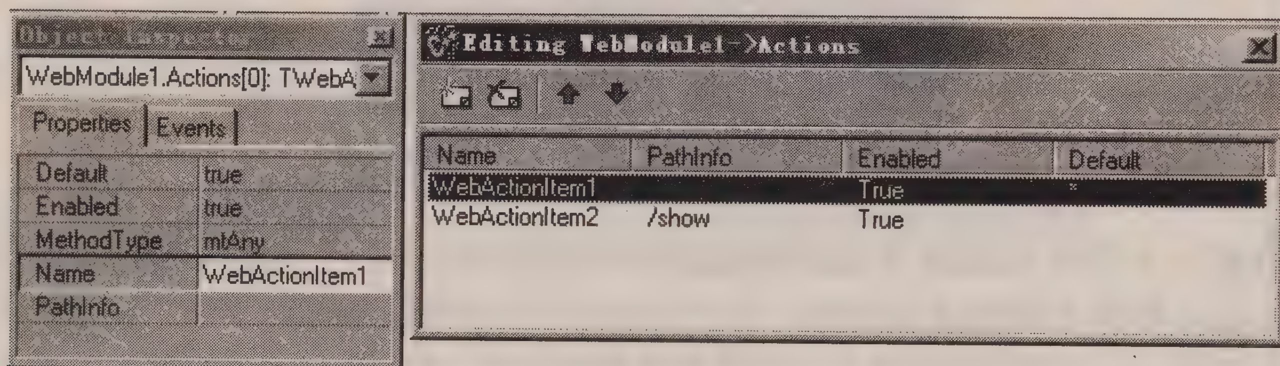


图 11-8　Web 模块的 Actions 属性编辑器

在 C++ Builder 中每个请求都被看作一个独立的事件，即 TWebActionItem 对象的 OnAction 事件。OnAction 事件处理函数用来对相应的动作项给出响应。该事件处理函数传递两个非常重要的参数 Request 和 Response。

Request 参数是 TWebRequest 类型变量，该类储存了用户请求的详细内容，其属性包括 Accept、Authorization、Content、ContentEncoding、ContentFields、ContentLength、ContentType、ContentVersion、Cookie、CookieFields、Date、From、Host、IfModifiedSince、Method、MethodType、PathInfo、PathTranslated、ProtocolVersion、Query、QueryFields、Referer、RemoteAddr、RemoteHost、ScriptName、ServerPort、Title、URL、UserAgent 等。从这些属性的命名上可以了解到，TWebRequest 类的属性与服务器传给 CGI 应用程序的环境变量相对应，从而可以简单的通过访问 Request 参数来得到所有必要的信息。

Response 参数是 TWebReponse 类型变量，TWebReponse 类的关键属性是 Content，可以通过该属性返回对指定请求的响应，即输出的 HTML 格式代码。对 Content 属性的赋值代替了在普通 CGI 应用程序中对标准输出设备输出文本。

在明确了以上关于 C++ Builder 的 WebModule 模式编程的基本概念后，可以开始着手编写具体程序了。创建一个没有 PathInfo 的动作项，并在改动作项 OnAction 事件响应函数中加入如下代码：

```
Response->Content=
    AnsiString("<HTML><HEAD><TITLE>Hello</TITLE>")+
    "</HEAD><BODY><h2>Hello(From CGI Application)</h2>"+
    "<hr>The Current Time is "+
    DateTimeToStr(Now()).c_str()+"</BODY></HTML>";
```

然后编译这个程序，一个简单的 CGI 应用程序就完成了。这个程序与 11.2.1 中创建的 CGI 程序作用相同，但却显得简明并且可读性好得多。这归功于 C++ Builder 对服务器应用程序的出色封装，和引入的可视化及事件委托编程模式。

11.2.3 实现网站计数器程序

CGI 的一个典型应用就是建立网站计数器程序，网站计数器目前在 Internet 上被广泛使用，几乎没有哪一个站点会不使用计数器（Hit Count）。在此将讲述关于编写图形界面的计数器应用程序方法，这涉及到一项重要的网络编程技术——向客户端输出图形数据。

创建这个程序的前期工作是准备几组图像数字。在建立的范例程序中，加入了如图 11-9 所示的五组数字图像。

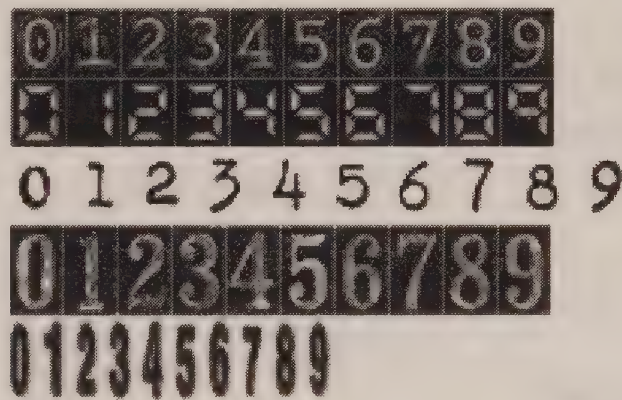


图 11-9 准备用于图形计数器的数字位图

将这五组图像作成资源文件 Num.res 加入到 Count 工程中，这五个图像分别命名为 NUM1~NUM5。WebCount 程序允许使用者通过参数指定使用哪种风格的数字显示。

该程序的基本原理很简单，将 Count 值保存在一个文本文件中，每当调用计数器程序时，将这个值显示出来，并递增该值后保存。下面所面临的棘手问题是，如何用图像的方式漂亮地将计数器显示出来。

向客户输出图像数据，其先决条件是必须设定生成描述为“image/jpeg”、“image/gif”或“image/bitmap”等。在 C++ Builder 中，可以通过访问 Response 参数的 ContentType 属性完成生成描述的设定。

另一方面，输出图像数据必须以流（Stream）的方式传递。根据上述两点，总结图形计数器的实现步骤如下：

(1) 打开保存 Count 值的文件，并读取该值。
(2) 根据调用参数（确定数字风格）和 Count 值生成图像。
(3) 将图像处理成 JPEG 格式图形。
(4) 声明一个内存流，并将 JPEG 图像数据存储在该流中。
(5) 通过 Response 参数的 ContentStream 和 ContentType 属性及 Send 方法传送图像数据。
(6) 递增 Count 值，并存回文件。

以下是网站计数器程序的关键代码。

Source 创建图形界面的网页计数器

```
//---------------------------------------------------------------------------
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
     int HitCount;
     int NH,NW; //NumHeight,NumWidth;
     char CountStr[7];
     FILE *fp;
     AnsiString FileName="c:\\WebCount.log";
     if (FileExists(FileName)){
          fp=fopen(FileName.c_str(),"r");
          fscanf(fp,"%ld",&HitCount);
          fclose(fp);
     }
     else
          HitCount=0;
     fp=fopen(FileName.c_str(),"w");
     HitCount++;
     fprintf(fp,"%ld",HitCount);
     fclose(fp);
     sprintf(CountStr,"%7ld",HitCount);
     Graphics::TBitmap *tmpBmp=new Graphics::TBitmap();
     Graphics::TBitmap *Bmp=new Graphics::TBitmap();
     TJPEGImage *Jpeg=new TJPEGImage();
     tmpBmp->LoadFromResourceName(0,"NUMS"+Request->Query);
     NH=tmpBmp->Height;
     NW=tmpBmp->Width/10;
     Bmp->Height=NH;
     Bmp->Width=NW*7;
     for (int i=0;i<7;i++){
          if (CountStr[i]==' ')
               Bmp->Canvas->CopyRect(Rect(i*NW,0,i*NW+NW,NH),
                    tmpBmp->Canvas,Rect(0,0,NW,NH));
          else
               Bmp->Canvas->CopyRect(Rect(i*NW,0,i*NW+NW,NH),
                    tmpBmp->Canvas,
                    Rect((CountStr[i]-'0')*NW,0,(CountStr[i]-'0'+1)*NW,NH));
```

```
        }
        delete tmpBmp;
        Jpeg->CompressionQuality=50;
        Jpeg->Assign(Bmp);
        delete Bmp;
        TMemoryStream *tmpStream=new TMemoryStream();
        Jpeg->SaveToStream(tmpStream);
        tmpStream->Position=0;
        Response->ContentStream=tmpStream;
        Response->ContentType="image/jpeg";
        Response->SendResponse();
        delete tmpStream;
        delete Jpeg;
}
//---------------------------------------------------------------------------
```

程序中 Request 参数的 Qeury 属性用于获取调用 CGI 程序的 URL 问号后面的那部分参数。在本程序中，此参数决定图形计数器的样式。

在 HTML 文档中加入类似下面所示的 HTML 标记，即可在网页中使用计数器程序：

```
<img src="cgi-bin/Count.exe?2" border="0" alt="计数器">
```

实际输出效果如图 11-10 所示。

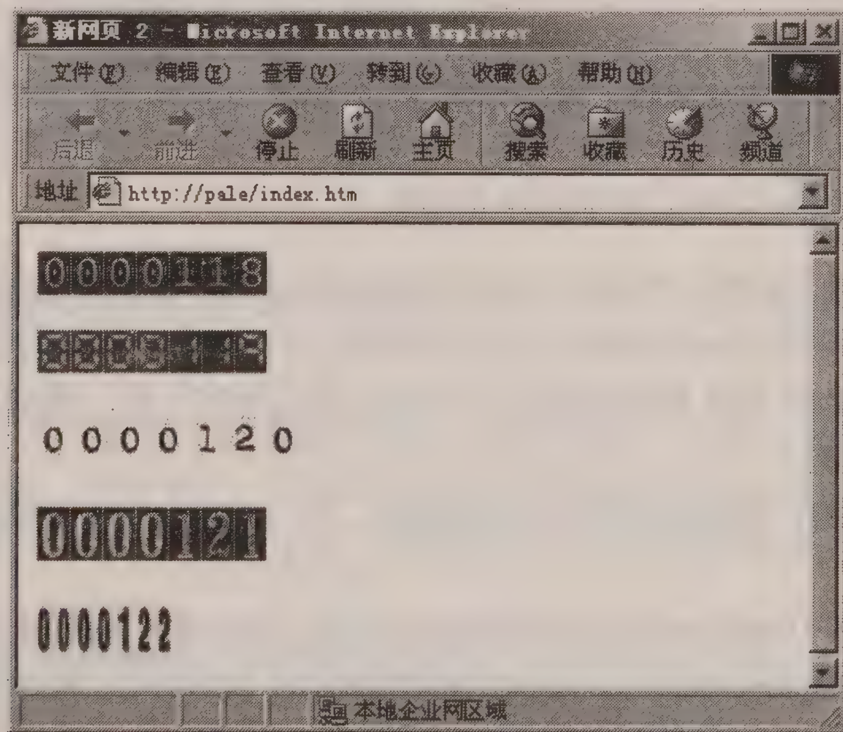


图 11-10 计数器的实际效果

11.3 创建基于 ISAPI 的留言簿系统

留言簿（GuestBook）也是服务器端程序的典型应用。一般留言簿系统需要有数据库支持，同时留言簿系统包括输入和输出两部分，涉及到使用窗体从客户机获取信息、与用户交

互、数据库显示等内容，比较有代表性。本章通过实现一个网页留言簿系统，具体讲述服务器端 Web 编程的一些问题。

11.3.1 ISAPI 编程概述

CGI 标准非常不错，目前仍然在 Internet 上被广泛使用。但现在几种 CGI 的替代方案已经日臻成熟，其中的娇娇者是 ASP（Active Server Page）和 ISAPI（Internet Server API）。

ASP 利用了某种脚本化语言来完成 CGI 所能完成的任务，它与 HTML 文档结合较为紧密，并且使用起来非常方便，其语言基础是 VBScript 或 JavaScript 等解释性宏语言，因此效率较低。

另一种目前已成为标准的技术是 ISAPI。这种技术使用一些专有的 API，使服务器接受客户编写的 DLL 程序，并将其装入常驻内存。在装入 DLL 后，服务器可以通过主进程中的线程执行单个请求，不像在 CGI 应用程序那样，必须重新启动一个 EXE。

ISAPI 在效率方面有明显优势，装载后的速度要比 CGI 和 ASP 高很多。服务器端应用程序与一般应用程序有一点本质不同，服务器端应用程序在同一时间被执行多次是司空见惯的事。比如说，同时有 5 个用户访问站点，程序就要被执行 5 次，如果是 CGI 程序的话，内存中就有这个进程的 5 个拷贝，而对于大的站点，同一时间有几百人乃至上千人访问是很常见的。可想而知，如果使用 CGI 程序将会造成多大内存空间和处理时间的浪费。另一点，对于 ISAPI 来说将用户请求内容发送到程序的过程完全发生在内存中，这一特性进一步加快了 ISAPI 应用程序的执行速度。



NSAPI（Netscape Server API）与 ISAPI 非常相似，它被 Netscape 的服务器系统所支持，完全可以认为是 ISAPI 的 Netscape 版本。事实上，NSAPI 更早出现，但 ISAPI 已经成为现实的标准，现在 Netscape 公司也宣布支持 ISAPI。

基于上述理由，我们将使用 ISAPI 创建网页留言簿系统。以通常方式实现 ISAPI DLL 具有相当的复杂性，但如果用 C++ Builder 开发 ISAPI 就不会遇到这样的问题。

在使用 C++ Builder 的 Web 框架具体开发 ISAPI 应用程序之前，我们首先来概括的了解一下 ISAPI DLL 程序的结构。

ISAPI DLL 与普通的 Windows DLL 基本相同。但 ISAPI 中必须实现三个特殊的入口函数，分别是：

```
BOOL __export WINAPI GetExtensionVersion(Isapi2::THSE_VERSION_INFO &Ver)
int __export WINAPI HttpExtensionProc(Isapi2::TEXTENSION_CONTROL_BLOCK &ECB)
BOOL __export WINAPI TerminateExtension(int dwFlags)
```

GetExtensionVersion 是最基本的入口函数，当服务器尝试将 DLL 调入内存时，将调用该函数。该函数可以确保 DLL 遵从服务器本身所遵守的规范。

TerminateExtension 函数在 DLL 卸载时调用。

HttpExtensionProc 函数是 DLL 的实际入口点，当每次执行 DLL 请求出现时，系统都会调用该函数。该函数直接接受一个参数，该参数是一个 TEXTENSION_CONTROL_BLOCK 结构，并且包含了很多有用的信息。这个结构定义如下：

```
struct TEXTENSION_CONTROL_BLOCK
{
    unsigned cbSize;
    unsigned dwVersion;
    unsigned ConnID;
    unsigned dwHttpStatusCode;
    char lpszLogData[80];
    char *lpszMethod;
    char *lpszQueryString;
    char *lpszPathInfo;
    char *lpszPathTranslated;
    unsigned cbTotalBytes;
    unsigned cbAvailable;
    void *lpbData;
    char *lpszContentType;
    TGetServerVariableProc GetServerVariable;
    TWriteClientProc WriteClient;
    TReadClientProc ReadClient;
    TServerSupportFunctionProc ServerSupportFunction;
};
```

该结构包含了四个回调函数的地址，其中 GetServerVariable 和 ServerSupportFunction 非常有用。GetServerVariable 函数用来返回服务器的连接信息或特定服务器的详细情况，这样就能够访问前面讨论过的 CGI 环境变量。ServerSupportFunction 函数可以访问一些服务器通用目的函数。

11.3.2 在 C++ Builder 中创建 ISAPI DLL

C++ Builder 提供了一个 Web 应用程序框架，这样消除了 CGI、Win-CGI、ISAPI、NSAPI 在程序编制方面的区别。具体来说，是生成 CGI 应用还是 ISAPI 应用，仅仅取决于创建 Web Application 时，选择了 CGI 项还是选择了 ISAPI 项，而具体的代码编写工作并没有什么不同。

如果在 New Web Application 对话框中选择了 ISAPI/NSAPI 项，C++ Builder 会自动生成 ISAPI/NSAPI 的程序框架，并提供与在 C++ Builder 中创建 CGI 程序相同的编程模式（WebModule）。C++ Builder 已经自动完成了 GetExtensionVersion、TerminateExtension、HttpExtensionProc 三个必要函数。打开工程的主入口文件，将会看到这三个函数的实现，如下所示：

```
BOOL __export WINAPI GetExtensionVersion(Isapi2::THSE_VERSION_INFO &Ver)
{
    return Isapiapp::GetExtensionVersion(Ver);
```

```
}
int __export WINAPI HttpExtensionProc(Isapi2::TEXTENSION_CONTROL_BLOCK &ECB)
{
    return Isapiapp::HttpExtensionProc(ECB);
}
BOOL __export WINAPI TerminateExtension(int dwFlags)
{
    return Isapiapp::TerminateExtension(dwFlags);
}
```

虽然 C++ Builder 封装了 ISAPI 的实现细节，但由于 ISAPI 应用本身的特殊性，创建 ISAPI 应用程序仍然有许多需要注意的地方。ISAPI 应用程序必须经过严格的调试和测试。由于 ISAPI 常驻内存，并且与服务器密切相关，如果 ISAPI DLL 出现故障或者一个微小的内存泄漏都会导致整个服务器崩溃。另一个问题是，ISAPI DLL 的调试较为困难，如果真正想看到 ISAPI 程序的运行结果，必须使 ISAPI DLL 被服务器装载。但是，一旦装载了 ISAPI DLL，则不能简单的替换 cgi-bin 或 Scripts 目录下的 ISAPI DLL 程序，如果想替换这个 DLL，唯一能做的是关掉整个服务器。

11.4 实现留言簿填写模块

整个留言簿系统包含两个模块：留言簿填写模块和留言簿浏览模块。这两个模块分别由两个不同的 ISAPI DLL 实现（GuestBook.dll 和 Write.dll）。本节将完成留言簿填写部分。

11.4.1 获取用户输入信息

HTML 不仅仅可以标记图像和文本，HTML 中还包含了一组标记用于产生一些窗体元素，这些窗体元素显示在 Web 页面上，它们通常用于接受用户输入的信息。

HTML 窗体元素包括：按钮、单行文本框、多行文本框、复选框、单选钮和密码输入框等。表 11-4 显示了窗体元素和与之对应的标记。

表 11-4 HTML 窗体元素和对应标记

名称	标 记
按钮	提交按钮：<input type="submit"> 重置按钮：<input type="reset"> 定制按钮：<input type="image">
文本输入框	普通文本框：<input type="text"> 密码输入框：<input type="password">
单选钮	<input type="radio">
复选框	<input type="checkbox">
组合框或列表框	<option>根据 height 值的不同决定显示组合框还是列表框
多行文本框	<textarea>

这些 HTML 中的窗体元素可以由<form>标记组合在一起。对于服务器端应用程序来说，<form>标记至关重要，<form>标记含有两个特殊的字段，action 字段指定发送数据的 Web 应用程序，method 字段指定用户数据的发送方式。同时，<form>标记范围内还应包含一个提交按钮，当用户单击此按钮后，所填写的信息被发送到服务器。

作为例子，我们来看看留言簿范例中用于用户输入数据的 HTML 文档，这一部分代码生成了图 11-11 显示的 HTML 窗体。

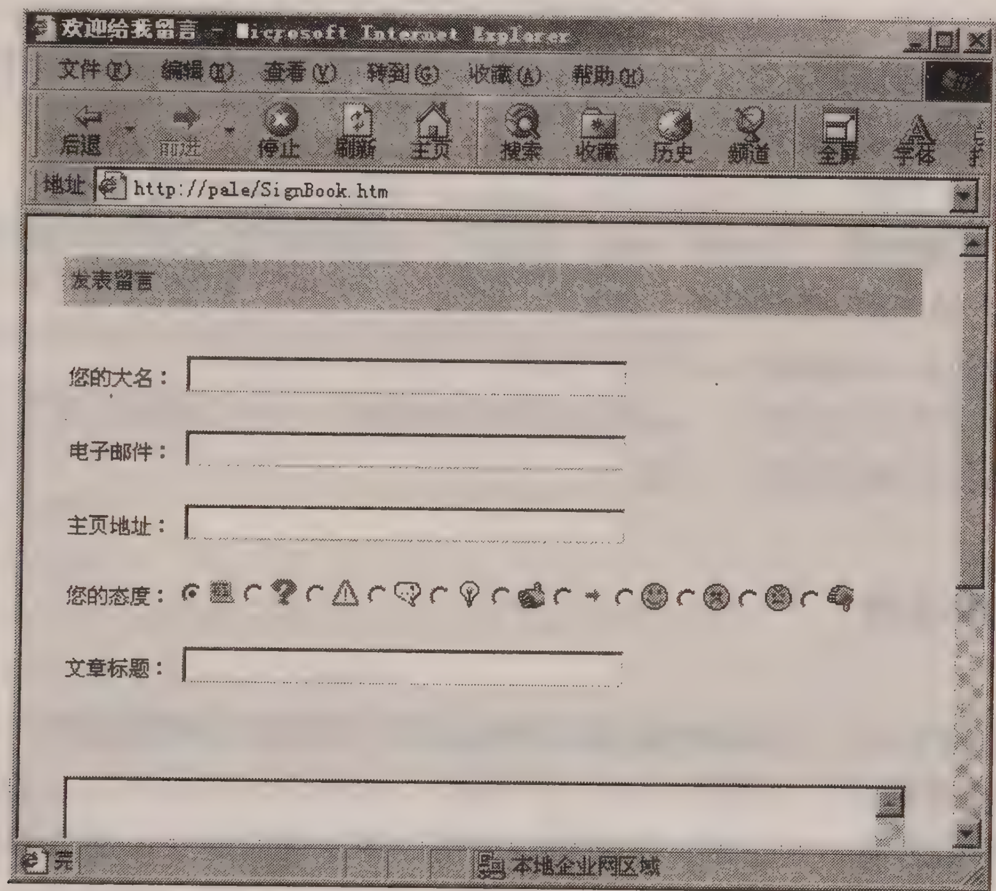


图 11-11 用于留言簿填写的 HTML 窗体

```
<form method="POST" action="cgi-bin/Write.dll">
<table><tr>
............
<td nowrap height="189">你的大名： <input type="text" maxLength="100" name="NAME" size="34">
<p>电子邮件：<input type="text" maxlength="100" name="EMAIL" size="34"></p>
<p>主页地址：<input type="text" maxlength="100" name="HOMEPAGE" size="34">
</p>
<p>你的态度： <input type="checked" name="EMOTION" type="radio" value="0"><img height="15"
src="images\mood0.gif" width="15">
............
<textarea cols="65" name="MEMO" rows="12"></textarea>
<div align="center"><center>
<p><input border="0" height="25" name="I1" src="images\logo_write_02.gif" type="image"
width="122">
............
</form>
```

用户在窗体中输入数据，输入内容将传给 action 字段指定的名为 Write 的 ISAPI 应用程序，在 Write 程序的 OnAction 事件响应函数中包含获取用户输入信息的代码，它是通过访问 Request 参数的 ContentFields 属性完成的。例如下面的代码：

```
Emotion=Request->ContentFields->Values["EMAIL"];
```

HTML 文档的<form>标记的 method 字段有两种取值——POST 和 GET。对于本实例使用的 POST 方法来说，OnAction 事件传递的 Request 参数的 Content 属性包含了使用这种方法发送到 Web 服务器的信息。也就是说，包括了<form></form>之间的各个窗体元素中用户输入的信息。可以通过 ContentFields 属性访问分解后的每一个域，其中每个字符串的形式是 Name = Value，字符串之间用&符号隔开。

对于另一种方法 GET，可以通过访问 Request 参数的 Query 属性得到。这个属性对应了 QUERY_STRING 环境变量。在使用 GET 方法时，传递的信息会追加在调用服务器应用程序的 URL 中，这样会使 URL 变得很长，例如：

http://www.mySite.com/gallery.exe/show?name=dog&color=black

这时，Query 属性的值将为 name=dog&color=black。与 Content 属性类似，一般通过 QueryFields 属性访问分解后的每一个域。

11.4.2 与数据库连接

对于网页留言簿系统，应该创建数据库来纪录用户留言以及相关信息。关于 C++ Builder 数据库编程，很多书籍都作为重点介绍，本书将不再讲述这方面内容。在此，假设你已对 C++ Builder 数据库编程有一定了解，本节将根据留言簿系统的需要，建立支持数据库的 ISAPI 程序。

当接收一个用户请求后，留言簿填写模块中的工作可以分为如下几步：

(1) 打开并初始化留言簿数据库。

(2) 获取用户输入并将用户输入数据存入数据库。

(3) 制作 HTML 页反馈输入数据。

代码并不复杂，如下所示。

Source 留言簿系统的数据库支持

```
//---------------------------------------------------------------------------
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
      AnsiString FieldName[5]={"NAME","EMAIL","HOMEPAGE","TITLE","MEMO"};
      AnsiString OutPut[7];
      AnsiString tmpStr;
      Table1->TableName="c:\\SignBook.dbf";
```

```
        Table1->Open();
        Table1->Insert();
        for (int i=0;i<5;i++){
            if (Request->ContentFields->Values[FieldName[i]]=="")
                Table1->FieldByName(FieldName[i])->AsString="空";
            else
                Table1->FieldByName(FieldName[i])->AsString=
            Request->ContentFields->Values[FieldName[i]];
            OutPut[i]=Table1->FieldByName(FieldName[i])->AsString;
        }
        Table1->FieldByName("EMOTION")->AsString=
            Request->ContentFields->Values["EMOTION"];
        OutPut[6]=Table1->FieldByName("EMOTION")->AsString;
        Table1->FieldByName("DATE")->AsDateTime=Date();
        OutPut[5]=Table1->FieldByName("DATE")->AsString;
        Table1->Post();
        Table1->Close();
        tmpStr="<HTML><HEAD><TITLE>致谢</TITLE></HEAD>";
        tmpStr="<BODY bgColor=\"#f4faff\">";
        tmpStr+="<H3>感谢批评和建议！</H3><BR><BR><SMALL>";
        tmpStr+="<B>发表时间：</B><BR>"+OutPut[5]+"<BR><BR>";
        tmpStr+="<B>姓名：</B><BR>"+OutPut[0]+"<BR><BR>";
        tmpStr+="<B>态度: </B> <img src=\"..\\images\\mood"+
        OutPut[6]+".gif\"><BR><BR>";
        tmpStr+="<B>电子邮件地址：</B><BR>"+OutPut[1]+"<BR><BR>";
        tmpStr+="<B>主页 URL:</B><BR>"+OutPut[2]+"<BR><BR>";
        tmpStr+="<B>留言标题:</B><BR>"+OutPut[3]+"<BR><BR>";
        tmpStr+="<B>留言内容:</B><BR>"+OutPut[4]+"<BR><BR>";
        tmpStr+="</SMALL></BODY></HTML>";
        Response->Content=tmpStr;
    }
    //---------------------------------------------------------------------------
```

这里采用 C++ Builder 中的 Database Desktop 7 工具建立留言簿数据库。在 Database Desktop 7 中可以生成如 Paradox 或 dBase 等多种类型的数据库表。以 dBase for Windows 类型构建留言簿数据库，并为数据库表创建 Name、Emotion、Email、HomepageTilte、Memo、Date 六个域，存为 signbook.dbf，如图 11-12 所示。

在 WebModule 容器中，我们加入一个 TTable 数据库组件，上述代码中，我们首先初始化 Table 数据表，将创建的 signbook.dbf 数据库和 TTable 组件连接。然后打开数据表，调用 Insert 方法，准备加入数据项。

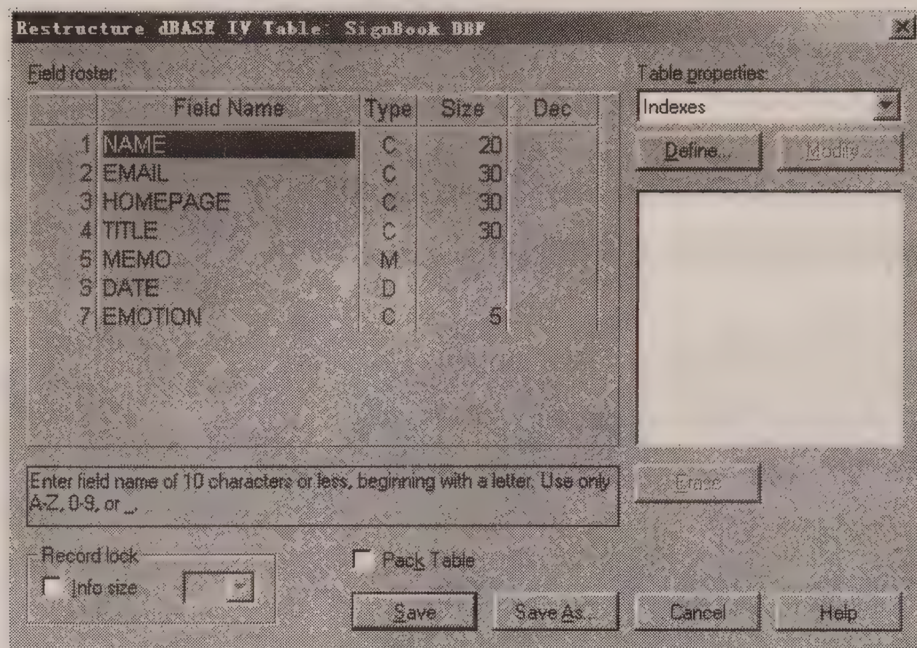


图 11-12　利用 Database Desktop 7 构建留言簿数据库

通过 Request 参数的 ContentFields 属性，获取用户输入信息，并将用户输入加入数据库。在 Output 数组中保存用户输入内容，准备创建 HTML 页反馈输入数据。

创建 HTML 显示页也很简单，只需将做好的 HTML 代码赋给 Response 参数的 Content 属性即可，其输出结果如图 11-13 所示。

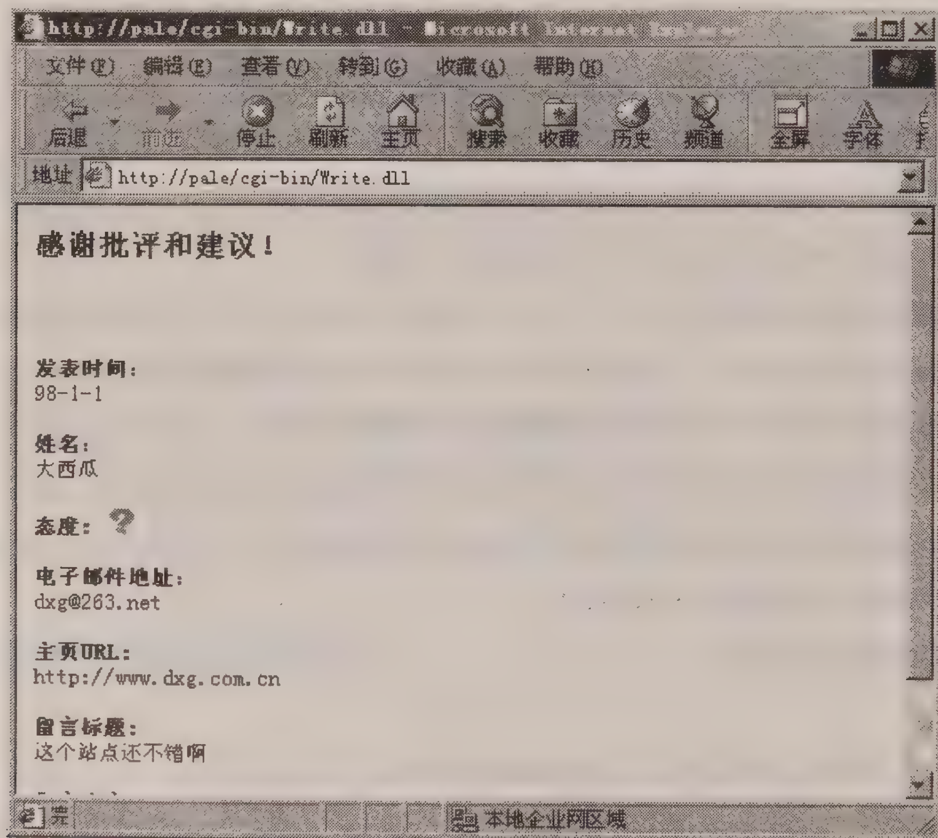


图 11-13　用户输入数据的显示页

11.5　实现留言簿浏览模块

浏览留言簿模块的主要任务是将留言簿数据库的内容公布出来，完全可以利用在第 1 小

节讲述的 DataSetTableProducer 组件快速显示数据库内容，这样做几乎不用输入什么代码。但用户往往希望给出更为漂亮的数据表格，这样 DataSetTableProducer 组件就不能满足要求了。对于本实例来说，使用 PageProducer 组件创建浏览留言簿的 HTML 文件。

11.5.1　显示留言列表

首先，使用 FrontPage、DreamWaver 等网页制作工具创建一个较为漂亮的 HTML 表格页面，表格中应填写留言簿内容的部分，用<#AddList>标记替代，实际留言簿数据部分在程序中生成，如图 11-14 所示。

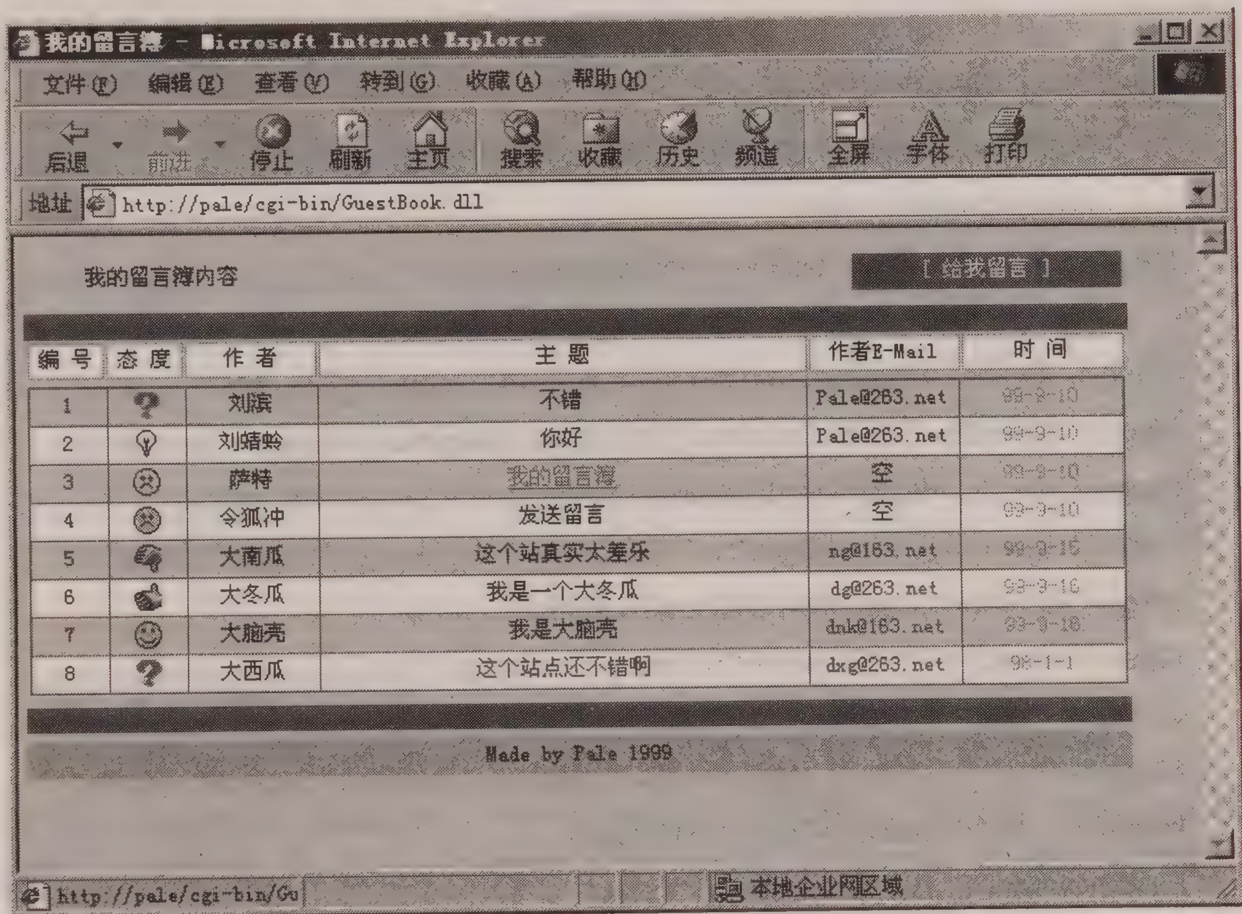


图 11-14　GuestBook.dll 程序生成的浏览留言簿界面

在 GuestBook 工程的 WebModul 容器中加入 PageProducer 组件，在 OnHTMLTag 事件响应函数中，包含生成留言簿内容列表的代码。

Source 实现留言簿内容列表生成

```
//---------------------------------------------------------------------------
void __fastcall TWebModule1::PageProducerHTMLTag(TObject *Sender,
    TTag Tag, const AnsiString TagString, TStrings *TagParams,
    AnsiString &ReplaceText)
{
        AnsiString Color1="\"#C2DEC6\"",Color2="\"#FFFFDF\"";
        AnsiString BGColor;
```

```
AnsiString Msg="";
Query1->Open();
Query1->First();
int Index=0;
if (TagString == "AddList"){
while (!Query1->Eof){
      Index++;
      if (Index%2==0)
            BGColor=Color2;
      else BGColor=Color1;
      Msg+="<tr>";
      Msg+="<td width=\"7%\" bgcolor="+
            BGColor+" height=\"20\" align=\"center\">"+I
            ntToStr(Index)+"</td>";
      Msg+="<td width=\"7%\" bgcolor="+BGColor+
            " height=\"20\" align=\"center\"><img src=\"..\\images\\mood"+
      Query1->Fields->FieldByName("EMOTION")->AsString+".gif\"></td>";
      Msg+="<td width=\"12%\" bgcolor="+BGColor+" height=\"20\" align=\"center\">";
      Msg+=Query1->Fields->FieldByName("NAME")->AsString+"</a></td>";
      Msg+="<td width=\"45%\" bgcolor="+BGColor+" height=\"20\" align=\"center\">";
      Msg+="<a href=\"GuestBook.dll/show?NAME="+
      Query1->Fields->FieldByName("NAME")->AsString+"\">"+
      Query1->Fields->FieldByName("TITLE")->AsString+"</a></td>";
      Msg+="<td width=\"14%\" bgcolor="+BGColor+" height=\"20\" align=\"center\">";
      Msg+="<a href=\"mailto:"+Query1->Fields->
      FieldByName("EMAIL")->AsString+"\">"+
            Query1->Fields->FieldByName("EMAIL")->AsString+"</td>";
      Msg+="<td width=\"19%\" bgcolor="+BGColor+" height=\"20\" align=\"center\">";
      Msg+="<font color=\"#50B14E\">"+
            Query1->Fields->FieldByName("DATE")->AsString+"</font></td>";
      Msg+="</tr>\n";
      Query1->Next();
   }
   ReplaceText=Msg;
   }
}
//-----------------------------------------------------------------------
```

所生成的 HTML 代码包含了一些 HTML 表格标记，如<tr>、<td>等，这些 HTML 内容在此不做讨论。为了美观，使用了两种不同的表格行背景颜色，并交替使用。对于 TITLE 字段和 EMAIL 字段，生成了 HTML 链接信息。TITLE 字段链接到了 GuestBook.dll 的 show 路径名

（PathInfo），这是本程序中的另一部分内容，它创建另一个 HTML 页显示详细的留言内容。

在这段代码中，通过 TQuery 组件而非 TTable 组件访问数据库内容。这里用到 First、Next 方法和 Eof 属性在一个 while 结构中遍历并显示整个数据表，Eof 属性判断是否在数据表尾。TQuery 组件与 TTable 组件相似，但 TQuery 组件多了 SQL 属性可以对数据表执行 SQL 命令，这在创建详细内容显示页面中会用到。

在完成 HTML 文档的生成代码之后，创建一个无 PathInfo 的动作项，在 OnAction 响应函数中显示由 PageProducer 组件生成的 HTML 文档。如下所示：

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    // 使用 SQL 命令初始化数据表格
    Query1->SQL->Clear();
    Query1->SQL->Add("select * from SignBook.dbf");
    Query1->Open();
    // 显示 PageProducer 组件创建的 HTML 文档
    Response->Content=PageProducer->Content();
    Query1->Close();
}
```

11.5.2 显示留言簿详细内容

当用户单击了某条留言标题时，GuestBook 程序将创建一个新的 HTML 文件显示这条留言的详细内容。为此在 WebModule 中创建路径名为 Show 的动作项，完成这项功能。

首先，应在数据集合中查找用户选择的数据项，使用 SQL 命令完成这项工作是方便的。通过前面创建留言列表部分所生成的 HTML 代码，可以看到链接 GuestBook.dll 时不仅使用了路径名 show，同时传递了留言项的 Name 字段。根据 Name 字段的值可以完成查找。如下面代码所示：

```
void __fastcall TWebModule1::WebModule1WebActionItem2Action(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    Query1->SQL->Clear();
    Query1->SQL->Add("select * from SignBook.dbf where NAME=\""+
        Request->QueryFields->Values["NAME"]+"\"");
    Query1->Open();
    Response->Content=PageProducer1->Content();
    Query1->Close();
```

}

当查找到数据项后，通过另一个 PageProducer 组件 PageProducer1 创建详细显示页面。实现方法与前面生成留言列表基本相同，具体代码请参看配套光盘。执行结果如图 11-15 所示。

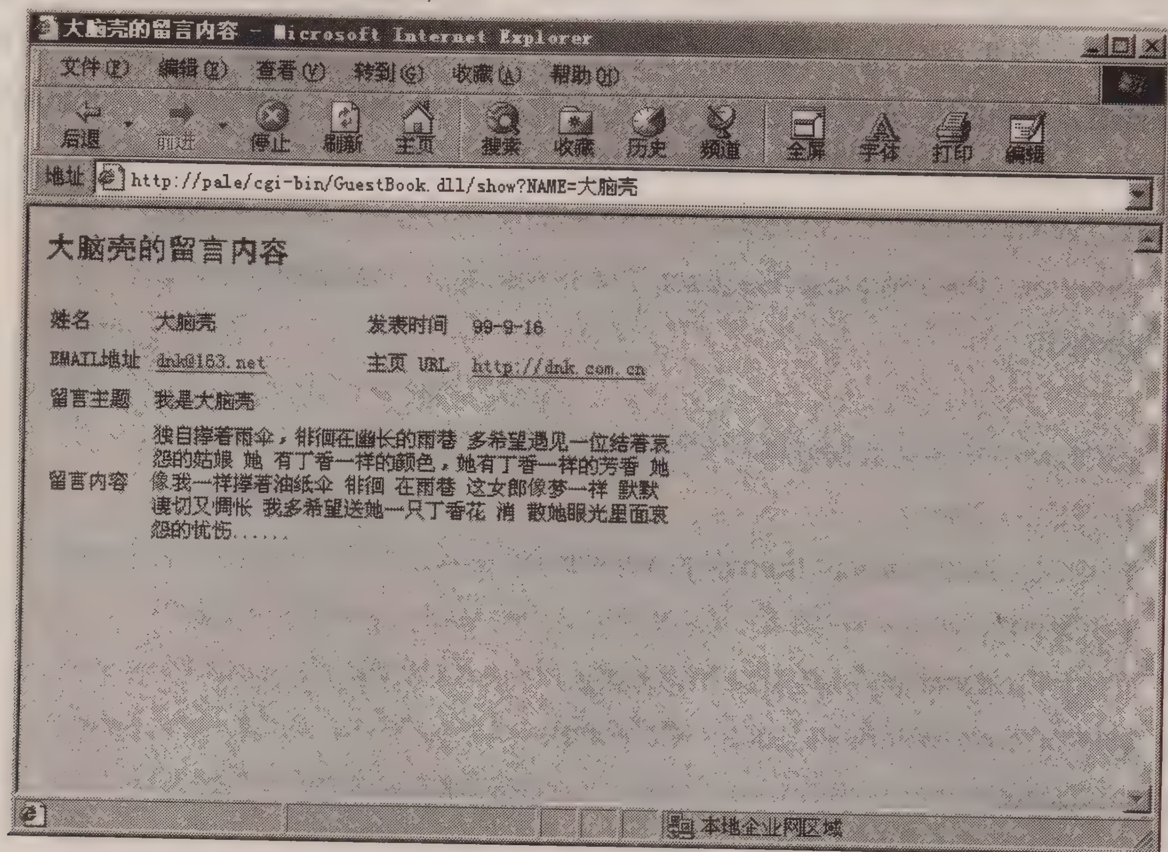


图 11-15　显示留言文章的详细内容

11.6　关于服务器端编程的进一步讨论

11.6.1　QueryTableProducer 组件

在本章的第 1 小节，我们讨论了 C++ Builder 中的三个页面生成器组件，包括 TPageProducer、TDataSetPageProducer、TDataSetTableProducer。实际上，C++ Builder 还提供了一个 TQueryTableProducer 组件，这个组件同样非常具有现实意义。它用于通过用户请求查询数据库，并将查询结果生成 HTML 格式表格显示出来。

对数据库的查询实际上还是通过 BDE 的查询引擎进行的，因此，TQueryTableProducer 组件必须与一个 TQuery 组件相连，TQueryTableProducer 组件包括一个 Query 属性实现这一点。需要将 TQuery 组件放到 Web 模块上，通过设置 DatabaseName 属性指定要访问的实际数据库，设置它的 SQL 属性指定要执行的 SQL 查询语句。查询结果将以 HTML 表格形式显示出来，表格生成过程与 TDataSetTableProducer 组件完全相同。

TQueryTableProducer 组件能够自动从 HTTP 请求消息中检索 URL 所附带的参数，如果客户的请求类型是 GET，通过 TWebRequest 对象的 QueryFields 特性可以得到 URL 中附带的参

数，如果客户的请求类型是 POST，通过 TWebRequest 对象的 ContentFields 特性可以得到 URL 附带的参数。如果 URL 附带的某个参数的名称与 SQL 语句中某个参数的名称匹配，TQueryTableProducer 组件就把该参数的值赋给 SQL 语句中与之匹配的参数。

TQueryTableProducer 组件的属性、方法和事件与 TDataSetTableProducer 组件几乎完全相同，此处不再赘述。

11.6.2 在线考试/问卷系统

在线考试/问卷在 Internet 上相当流行。从技术角度来讲，在线考试/在线问卷系统并没有什么特别的地方。对于在线问卷系统来说，只是简单的搜集信息并存储在后端数据库，并做一些统计工作。而简单的在线考试程序甚至常常不涉及数据库。

对于在线考试/问卷程序，经常需要从各种 HTML 窗体元素中获得数据。不同的 HTML 窗体元素，设置和获得数据的方式有所不同。比较特殊的单选钮（radio）和复选框（checkbox）。

在 HTML 构建一组单选钮（radio）必须使每一个单选钮的 Name 字段相同，并且设定每一个单选钮的 Value 字段为不同值。例如，单选题 A/B/C/D 四项分别可以设定 Value 值为"A"、"B"、"C"、"D"。在程序中，通过 ContentFields 或者 QueryFields 访问这组单选钮（因为有共同的 Name），得到的值将是用户选择的那一项的单选钮元素的 Value 值。

复选框在 HTML 中创建有些不同，一组复选框中每一个复选框的 Name 字段必须不

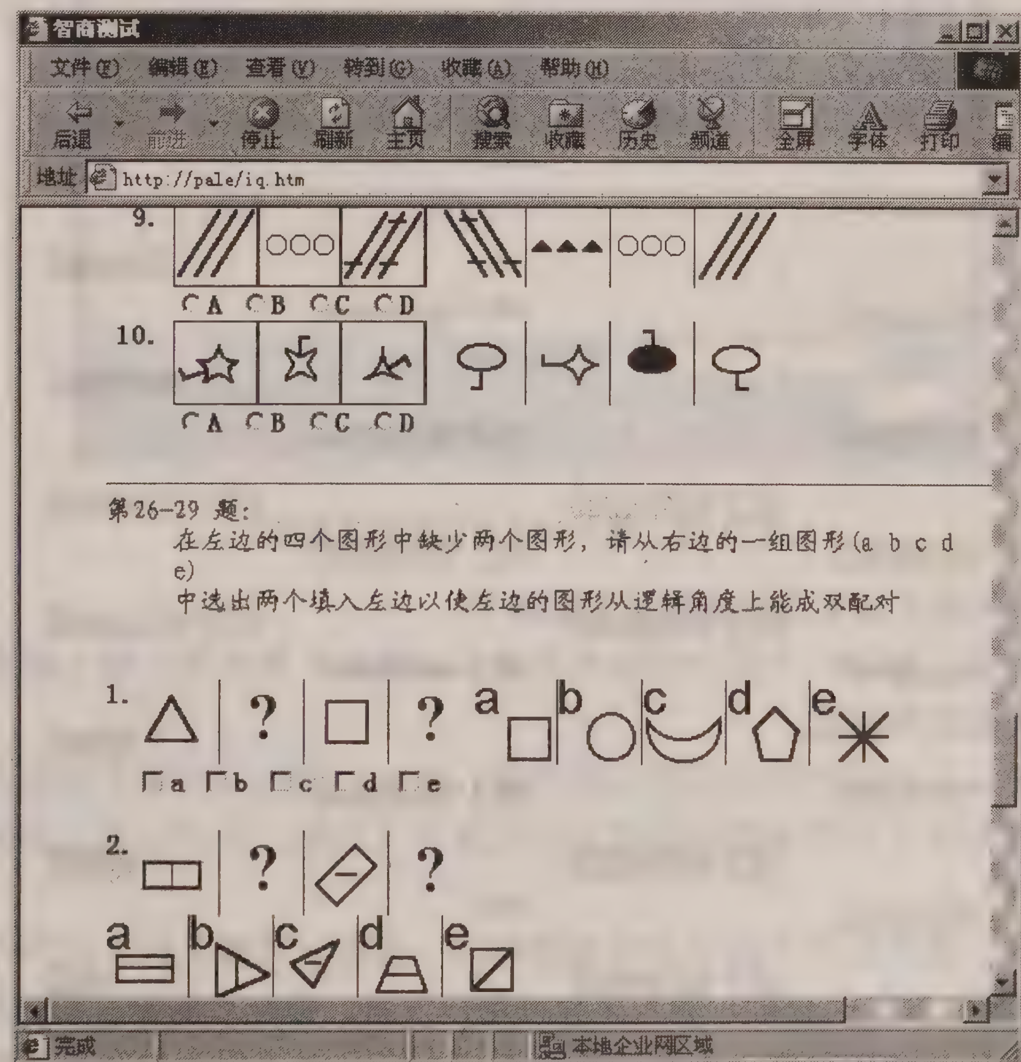


图 11-16 在线智商测试程序

同，同时设定 Value 字段。当某一个复选框被用户选中时，在 Request 参数的 Query 属性或者 Content 属性中将包含 Name=Value 这样的内容。

在配套光盘上，你可以看到一个智商测验的 ISAPI 程序，如图 11-16 所示。



常常看到网上一些考试/测验程序以 JavaScript 实现，这确实非常简单，并且表面上看好像没有什么问题，但这种做法是不合适的，因为对 HTML 和 JavaScript 稍有了解的用户可以简单地选择“查看源文件”来获得题目的全部答案。

11.6.3 聊天室系统

聊天室在 Internet 上同样非常常见，无疑利用 ASP 创建聊天室系统是最为简单而有效的。但创建 CGI 或者 ISAPI 程序也能完成同样的任务。图 11-17 给出了一个基于 Web 页的聊天室程序。

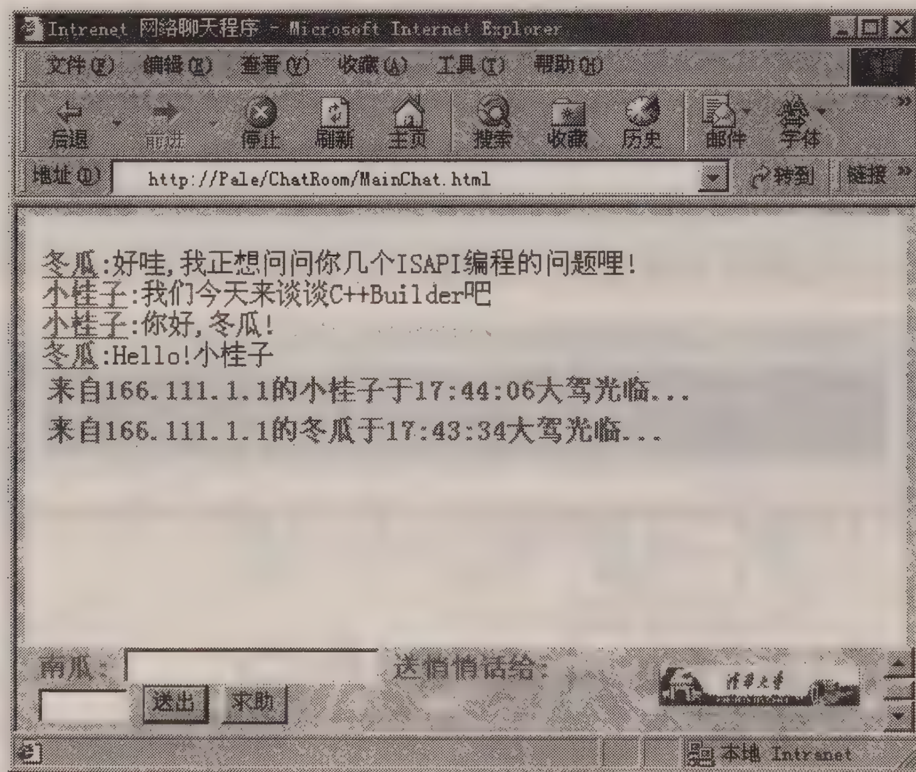


图 11-17　基于 Web 页的聊天室程序

创建一个典型的聊天室系统应该包含下面部分：

- 聊天室首页。包含一个用户登陆和身份认证程序，要求用户输入匿名或密码。该程序与用户数据库相连。
- 通过身份认证的用户被允许进入聊天室。同时，该用户的名字被加入当前用户名列表。这个列表可以简单的存储在文件记录中（当然，也可以通过其他方式进行进程间的通信）。
- 聊天页面。通常这个页面至少包含两部分：信息区和输入区。这两个部分（页面）通过 HTML 的框架分割，输入区允许用户发言，提交后的发言将被加入一个队列结构中。该队列结构存储了近期的所有用户发言，对于保存的发言条数有程序控制。队列结构中的信息也保存在文件记录中（或使用内存映像文件，见第 15 章），信息

区从文件读近期的取用户发言信息，并转换为 HTML 文档显示出来。为了及时显示最新的发言，该页面必须自动更新。可以通过 HTML 文档中的<meta>标记实现该页的定时自动更新。例如：

```
<meta http-equiv="Refresh" content="10,http://pale/cgi-bin/message.dll">
```

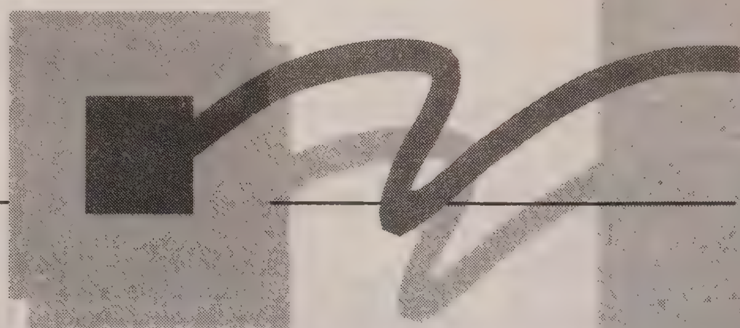
这行代码可以使页面每 10 秒刷新一次。

- 用户退出聊天室，显示用户退出信息，并将该用户从用户列表中删去。

第 12 章

网络五子棋
——WinSock 编程

TClientSocket 和 TServerSocket 组件
创建网络五子棋实例
多线程连接、流式传输
定制协议
PowerSock 组件



本章导读

从程序员的角度看，WinSock 是一组用 C 语言写的 API，用于通过 Internet 传输数据。虽然现在 C++ Builder 有很多现成组件可以实现通过 FTP、HTTP 等高级协议在 Internet 上传输数据和文件，但是通过 WinSock 编程可以获得更大的灵活性。

本章将创建可联网对战的游戏程序——网络五子棋。漂亮的界面、稳定的联机对战性能、丰富的辅助功能是该范例程序的特色。通过这个程序的实现，讲述 C++ Builder 中应用 WinSocket 组件——TClientSocket 组件和 TServerSocket 组件的编程技术。本章同时涉及 WinSock 编程的一些高级问题，以及 C++ Builder 的 WinSock 组件替代技术——更为强劲的 PowerSock 组件。

本章内容包括以下方面:

- 在 C++ Builder 中使用 WinSock 组件 TClientSocket 和 TServerSocket。
- 应用 WinSock 组件创建实例程序网络五子棋。
- 关于 WinSock 编程的一些高级话题，包括多线程连接、流式传输、定制协议等。
- 强劲的 WinSock 组件替代技术——PowerSock 组件。

12.1 WinSock 编程概述

早在 Windows 出现之前，Internet 就已经存在了。时至今日，它已经成为世界上最大的 TCP/IP 网络。在 UNIX 机上，一组称为 Berkeley Sockets 的约定成为 Internet 上 UNIX 机通过 TCP/IP 协议通信的标准。其他操作系统也都实现了 TCP/IP 通信，这极大地促进了 Internet 的发展。但由于这些不同的操作系统对 TCP/IP 的实现有所不同，使事情变得混乱。于是，一些著名的厂商联合起来制定了 WinSock 规范。

最初，进行 WinSock 编程意味着直接调用 WinSock API 函数。这组函数一般包含在 WinSock.dll 或 Wsock32.dll 中。低级的 WinSock API 提供了建立 TCP/IP 连接所需的所有支持。同时也支持其他相关的协议，如 XNS、DECnet 以及 IPX/SPX 等。

Internet 的核心是 TCP/IP（Transmission Control Protocol/Internet Protocol）。这两种协议组合使用提供了 Internet 上的连接，也包括在 Intranet 上的连接。更进一步说，IP 负责定义 Internet 传输单元，并指定寻址方案；TCP 负责较高级别的传送服务。高级协议（包括 FTP，HTTP，SMTP，Telnet 等）都是建立在 TCP/IP 基础之上的。

直接与 WinSock API 打交道极为不便。在 C++ Builder 中提供了 TClientSocket 组件和 TServerSocket 组件封装了低级的 WinSock API 函数，这两个组件帮助用户方便的实现了 WinSock 编程。大多情况下，Socket 工作是基于连接的：两个程序通过 Socket 连接在其两端并通过连接接收和发送数据。而 TClientSocket 组件和 TServerSocket 组件分别用来操纵客户端 Socket 与服务器端 Socket 的连接和通信，要说明的是，这两个组件只用于管理服务器和客户的连接，对于具体的数据传输需要操纵实际的 Socket 对象，包括 TCustomWinSocket 及其派生类，如 TClientWinSocket、TServerWinSocket 和 TServerClientWinSocket 等。

位于 Internet 选项板上的 Winsock 组件如图 12-1 所示。

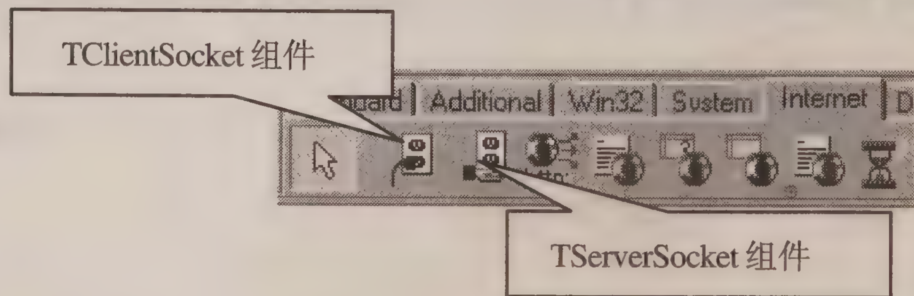


图 12-1 位于 Internet 选项板的 WinSock 组件

如何通过 WinSock 组件来启动 Socket 通信呢？服务器端应用程序会首先启动，它将等待来自客户机程序的请求，在另一台计算机上的客户机程序请求一个连接。为此，客户端的 Socket 必须首先描述它要连接的服务器端 Socket（主要是指服务器端 Socket 的地址和端口号），然后再定位所要连接的服务器端 Socket，找到以后，就向服务器端 Socket 请求连接。而后，服务器程序允许连接，即向客户端 Socket 发出“允许连接”（Accept）的信号，这时客户端 Socket 与服务器端 Socket 的连接就建立起来了。

12.1.1 建立服务器端 Socket

把一个 TServerSocket 组件加到应用程序窗体上，这个应用程序就变成了一个 TCP/IP 服务器。要使服务器能监听客户的连接请求，必须首先通过 TServerSocket 组件的 Port 属性指定端口号。当然，如果服务器提供的是某些标准的服务如 FTP、FINGER 或 TIME，这些标准服务的端口号是事先约定的，也就是说，可以通过设定 TServerSocket 组件的服务类型（通过 Service 属性），隐含确定端口号。如果同时设置了 Port 属性和 Service 属性，C++ Builder 将忽略 Port 属性。

这之后可以调用 TServerSocket 组件的 Open 对象方法或者将 Active 属性设为 true 启动服务器监听状态。这时，当客户机提出连接请求，服务器就自动接受请求建立连接并触发 OnClientConnect 事件。当连接建立完成后，可以通过访问 TServerSocket 的 Socket 属性进行与客户机的通信和数据传输工作。

在服务器端调用 TServerSocket 组件的 Close 方法或者把 Active 属性设为 false，将断开所有与客户的连接，并退出监听状态。

12.1.2 建立客户端 Socket

把一个 TClientSocket 组件加到应用程序窗体上，这个应用程序就变成了一个 TCP/IP 客户机。要建立与服务器的连接，必须首先通过 TClientSocket 组件的 Host 属性或 Address 属性指定要连接的服务器。Host 属性支持通过服务器主机名指定服务器，Address 属性用于指定服务器的 IP 地址。同时，指定连接的服务器的端口号也是必须的。与 TServerSocket 组件一致，可以通过 Service 属性或者 Port 属性设定连接服务器的端口号。

之后只要调用 Open 方法或者把 Active 属性设为 true，客户端 Socket 就向服务器端 Socket 提出连接请求。如果服务器此时处于监听状态，就会自动接受请求并建立连接。

建立连接会触发 OnConnect 事件。当连接建立完成后，可以通过访问 TClientSocket 的 Socket 属性进行与服务器的通信和数据传输工作。

调用 TClientSocket 组件的 Close 方法或者把 Active 属性设为 false，将断开该客户与服务器的连接。此时，服务器端将触发 OnClientDisconnect 事件，而客户端将触发 OnDisconnect 事件。

12.1.3 操纵 Socket 对象传输数据

TClientSocket 组件和 TServerSocket 组件用于管理服务器和客户的连接，对于具体的数据传输，需要操纵实际 Socket 对象（通过 Socket 属性访问），对于服务器端来说，Socket 对象为 TServerWinSocket 类的实例，客户端 Socket 对象则为 TClientWinSocket 类的实例。而 TServerWinSocket 类和 TClientWinSocket 类都继承于共同父类 TCustomWinSocket，该类包含了重要的通信和数据传输方法。

以下介绍 TCustomWinSocket 类、TServerWinSocket 类和 TClientWinSocket 类的重要属性、

方法和事件。

1. TCustomWinSocket 类的重要属性和方法

- Connected 属性。

声明：__property bool Connected = {read=FConnected, nodefault};

该属性描述 Socket 的连接状态，返回 true 表示处于连接状态。一般在使用 Socket 传输数据之前，首先要访问 Connected 属性，看看当前是否已经正常连接。

- LocalAddress/LocalHost/LocalPort 属性。

这三个属性分别返回本机 Socket 的 IP 地址、主机名和端口号。一个 IP 地址可以同时用于多个 Socket 连接，但每一个 Socket 连接的端口号必须是相异的。

- RemoteAddress/RemoteHost/RemotePort 属性。

这三个属性分别返回与本机连接的另一端 Socket 的 IP 地址、主机名和端口号。如果客户向服务器请求某种特定的服务，RemotePort 属性返回的端口号可能不同于客户请求的端口号。

- SocketHandle 属性。

声明：__property int SocketHandle = {read=FSocket, nodefault};

返回 Socket 对象的 Windows 句柄，在直接调用 WinSock 的 API 时需要用到这个句柄。如果连接还没有建立，SocketHandle 属性返回 INVALID_SOCKET。

- ReceiveBuf 方法。

声明：int __fastcall ReceiveBuf(void *Buf, int Count);

该函数用于从 Socket 连接中读取 Count 个字节到 Buf 参数中，并返回实际读取的字节数。ReceiveBuf 函数适用于非阻塞方式的 Socket。一般是在处理 Socket 对象的 OnSocketEvent 事件或者 TServerSocket 组件的 OnClientRead 事件或者 TClientSocket 元件的 OnRead 事件时调用。

- ReceiveLength 方法。

声明：int __fastcall ReceiveLength(void);

该函数用于估计从 Socket 连接中读取数据的字节数。

- ReceiveText 方法。

声明：AnsiString __fastcall ReceiveText();

该函数从 Socket 连接中读取一个字符串并返回这个字符串。

- SendBuf 方法。

声明：int __fastcall SendBuf(void *Buf, int Count);

使用 SendBuf 函数可以向另一端 Socket 发送数据。它将缓冲区 Buf 中 Count 个字节发送到 Socket 连接上，并返回实际发送的字节数。

- SendStream/SendStreamThenDrop 方法。

声明：bool __fastcall SendStream(Classes::TStream* AStream);

SendStream 函数向另一端 Socket 发送一个流式对象，如果发送成功，则返回 true。SendStreamThenDrop 函数与 SendStream 函数类似，但它将在流发送完毕后使 Socket 断开连接。

- SendText 方法。

声明：void __fastcall SendText(const AnsiString S);

该函数用于向另一端 Socket 发送一个字符串。

2. TServerWinSocket 类的重要属性、方法和事件

- ServerType 属性。

声明：__property TServerType ServerType = {read=FServerType, write=SetServerType, nodefault};

该属性用于设置与客户的连接方式，可取值为 stThreadBlocking 和 stNonBlocking。stThreadBlocking 表示与每一个客户的连接都自动分配一个线程(TServerClientThread 对象)；stNonBlocking 表示用非阻塞方式连接每一个客户。

- ActiveConnections 属性。

声明：__property int ActiveConnections = {read=GetActiveConnections, nodefault};

返回当前活动的连接数，实际上也就是 Connections 数组的元素个数。

- ActiveThreads 属性。

声明：__property int ActiveThreads = {read=GetActiveThreads, nodefault};

返回当前正在使用的 TServerClientThread 对象的个数，通过 TserverClientThread 可以管理与客户连接的每个线程。

- Connections 属性。

声明：__property TCustomWinSocket* Connections[int Index] ={read= GetConnections};

该属性数组的每一个元素都代表一个当前活动的连接(TServerClientWinSocket 对象)，通过该属性可以方便的获取每一个客户连接的详细信息。

- Listen 方法。

声明：HIDESBASE void __fastcall Listen(AnsiString &Name, AnsiString &Address, AnsiString &Service, Word Port, int QueueSize);

该方法使服务器端 Socket 进入监听状态。

- OnClientConnect 事件。

当服务器收到一个客户连接请求后，服务器 Socket 对象将自动创建 TServerClientThread 线程对象用于处理该用户连接。正常建立连接后，将触发 OnClientConnect 事件，通知服务器端应用程序 Socket 连接已完成。OnClientConnect 事件是服务器端程序了解相连的客户机的第一次机会。

- OnClientDisconnect 事件。

当服务器端 Socket 与某个客户的连接被断开时将触发这个事件。如果 ServerType 属性设为 stThreadBlocking，在该事件发生之后还将触发 OnThreadEnd 事件。

- OnClientRead 事件。

当客户机有数据发送过来时，将触发此事件通知服务器 Socket 读取有关信息。

- OnClientWrite 事件。

该事件通知服务器端 Socket 对象去写信息。

3. TClientWinSocket 类

TClientWinSocket 类直接继承于 TCustomWinSocket 类，不同之处仅仅是增加了一个

ClientTyoe 属性。

- ClientType 属性。

声明：enum TClientType { ctNonBlocking, ctBlocking };

__property TClientType ClientType = {read=FClientType, write=SetClientType, nodefault};

ClientType 属性用于设置阻塞方式的 Socket 和非阻塞方式的 Socket。如果把 ClientType 属性设为 ctNonBlocking（非阻塞方式），客户端 Socket 将进行异步读写，即在进行读或写的操作时，不影响其他线程的执行。如果把 ClientType 属性设为 ctBlocking（阻塞方式），读或写的操作必须在两端同步进行。在阻塞方式下，可以使用 TWinSocketStream 类进行数据传输。

12.2　实例创建分析

本章将创建一个具有简单连网对战功能的范例程序——网络五子棋。这个程序用于演示 C++ Builder 下的 WinSock 编程，它基于 C++ Builder 的 TServerSocket 组件和 TClientSocket 组件。

WinSock 编程基于服务器程序和客户机程序的连接，这就意味着一般需要分别实现服务器端程序和客户机端程序。但在本例中，将服务器端和客户机端程序合二为一。具体来说，在五子棋程序窗体上同时放入 TServerSocket 组件和 TClientSocket 组件，表示五子棋程序既可以作为 TCP/IP 服务器，也可以作为 TCP/IP 客户。五子棋程序如图 12-2 所示。

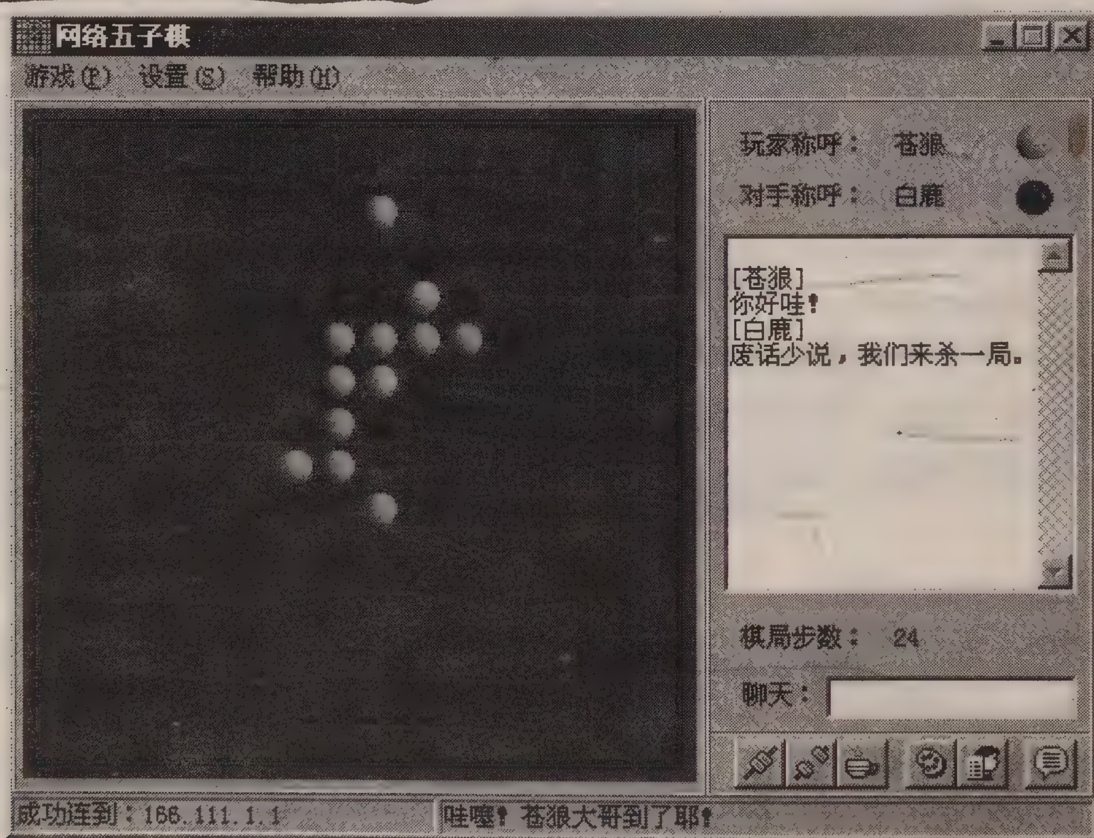


图 12-2　可以通过网络对战的五子棋程序

设计期间，将 TServerSocket 组件和 TClientSocket 组件的 Port 属性均设为 1024，以设置 1024 为本程序 TCP/IP 通信的端口号，当然也可设置其他空闲的端口号。

作为一个支持连网对战的程序，实现游戏双方间的数据传输是本程序需要完成的主要工

作。程序同时提供了聊天功能，这就要求考虑两类信息的传输：聊天文本信息和行棋信息。

同时，游戏系统的构建也是非常重要的。重点包括棋盘棋子的绘制，游戏状态的表示，游戏胜负判定等等。由于本例旨在演示 WinSock 编程，游戏系统在某些地方的处理较为简单。

具体来说，在网络五子棋程序的设计过程中，需要逐步完成以下任务：

- 设计并创建应用程序界面。
- 完成该程序的初始化工作。包括棋盘绘制，服务器和客户机的连接处理等。
- 实现游戏双方间的数据传输，创建完整的游戏系统，同时包括简单的聊天功能，这是整个程序的主体部分。
- 简单的辅助功能，包括设置棋盘颜色和设置棋盘背景。

通过实现该范例程序，你将能掌握在 C++ Builder 中使用 TServerSocket 和 TClientSocket 组件进行 WinSock 编程的方法和实用技巧。

该程序应该在两个不同的计算机上运行，这两台计算机必须有固定的 IP 地址。

其实，在单机上也可以调试这个程序。即使这台计算机并没有连入 Internet，只要计算机安装了 TCP/IP 协议（通过“控制面板”的“拨号网络”），并指定了 IP 地址，就可以在本机进行连接并调试。

单机调试时输入主机名或 IP 地址时，可以键入本机的 IP（例如 166.111.1.1）或者“localhost”。单机调试时需要注意的是，在启动第二个程序之前，应该将首先启动的程序的监听状态取消，否则会引起异常。

12.3　实现网络五子棋程序

在进行具体的编程之前，首要的任务是创建该程序的用户界面，如图 12-3 所示。

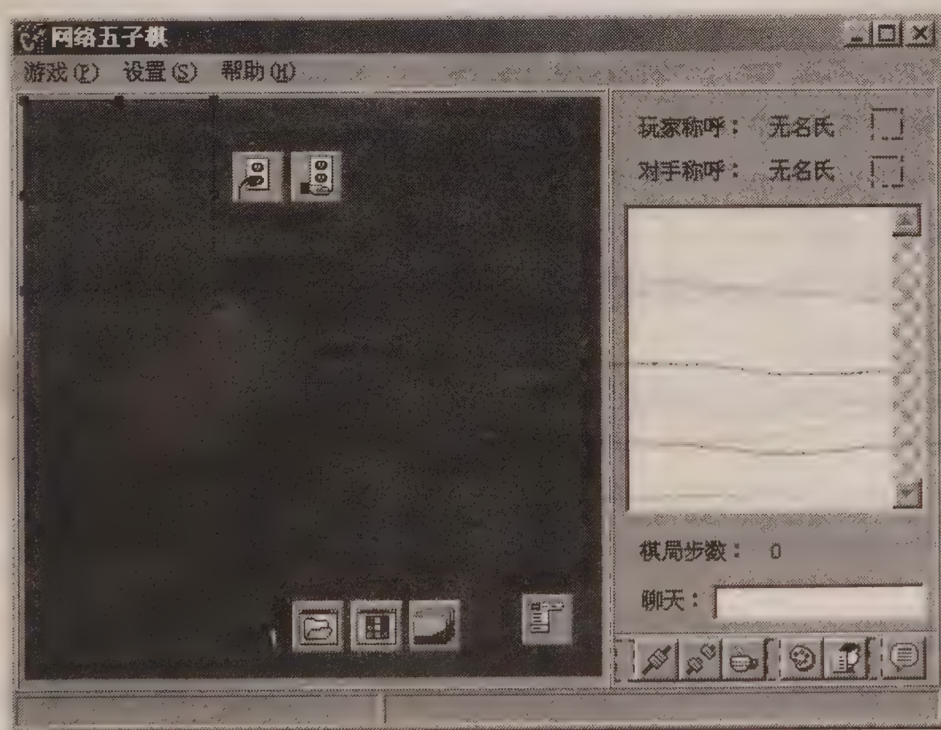


图 12-3　网络五子棋程序界面设计

窗体中放入的 TServerSocket 组件和 TClientSocket 组件是本程序的核心组件，用于对服务器 Socket 和客户机 Socket 连接和通信提供支持。棋盘部分包括两个 TImage 组件，组件 BKImage 提供棋盘背景图案，组件 Image 的 Transport 属性设定为 true，该组件覆盖在 BKImage 之上，用于绘制棋盘线和黑白棋子。

其他组件包括一个 TMemo 组件，用于显示聊天信息；在窗体右上方有两个 TLabel，用于显示自己和对手的匿名；两个 TLabel 组件右侧各有一个 TImage 组件用于显示自己和对手的棋子颜色；此外还有菜单、工具栏、状态栏等等组件，在此不再赘述。

12.3.1 游戏前期工作

在服务器端程序和客户机程序连接并进行游戏之前，有一些必要的前期工作需要完成。首先，在用户进入时应提示用户输入名字；其次，应该绘制棋盘线，这项工作在应用程序启动时完成；第三项任务与 WinSock 编程相关，应该对用户提出的请求作出响应，即进行服务器和客户机的连接处理。

1. 使用 InputQuery 函数提示输入

本程序要求一个匿名用于用户标识，请看主窗体的 FormCreate 函数。

Source 进入五子棋程序时的匿名输入

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
	while(!InputQuery("欢迎进入", "请输入你的匿名:", mName)||Name=="");
	Label2->Caption=mName;
	StatusBar1->Panels->Items[1]->Text = "哇噻！ "+
	mName+"大哥到了耶！ ";
	OpenDialog1->Filter = GraphicFilter(__classid(TGraphic));
	initialize(); // 初始化棋盘
	WaitItemClick(NULL); // 进入监听状态
}

//-----------------------------------------------------------------------
```

程序中使用了 InputQuery 函数提示用户输入匿名。该函数声明如下：

```
bool __fastcall InputQuery(const AnsiString ACaption, const AnsiString APrompt, AnsiString &Value);
```

这个函数用于产生一个标准输入对话框（如图 12-4 所示）。其中 ACaption 参数用于设定对话框的标题，APrompt 参数指定提示信息字符串，Value 参数用于返回用户输入内容。如果用户按下 OK，该函数返回 true，按下 Cancel 则返回 false。

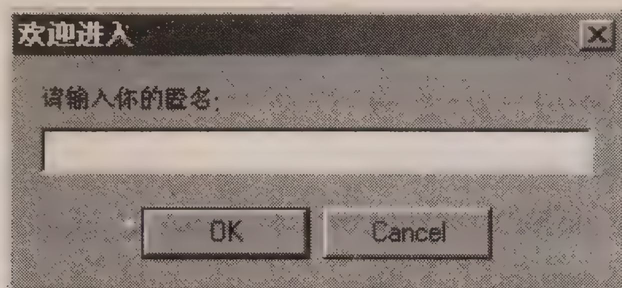


图 12-4　使用 InputQuery 函数产生的标准输入对话框

2. 初始化棋盘

在 FormCreate 函数中调用了创建的 initialize 函数，进行棋盘和棋局的初始化工作。该函数内容如下：

Source 初始化棋盘

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::initialize()
{
        Image->Picture->Graphic=NULL; // 擦出棋盘前景图像
        canPut=false;stepNum=0;    // 初始化步数
        Label7->Caption=IntToStr(stepNum);
        Image->Height=BKImage->Height;
        Image->Width=BKImage->Width;
        DrawLines(); // 绘制棋盘线
        for (int i=0;i<15;i++)
                for (int j=0;j<15;j++)
                        State[i][j]=sNone;
}
//---------------------------------------------------------------------------
```

在本程序中，定义 TState 类型用于表示棋盘上某点的状态，该类型定义如下：

```
enum TState {sWhite,sBlack,sNone};
```

同时，程序中定义一个 TState 类型数组 State[15][15]，在 initialize 函数中初始化这个数组。

另一方面，在 initialize 函数中，调用 DrawLines 函数进行棋盘线的绘制工作。此函数绘制了 15×15 的棋盘线并描绘了棋盘边框，函数内容如下：

Source 棋盘绘制实现

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::DrawLines()
```

```
{
        Image->Canvas->Pen->Width=1;
        for (int i=0;i<16;i++){    // 绘制内部棋盘线
              Image->Canvas->MoveTo(8+i*20,8);
              Image->Canvas->LineTo(8+i*20,308);
              Image->Canvas->MoveTo(8,8+i*20);
              Image->Canvas->LineTo(308,8+i*20);
        }
        // 绘制边框线
        Image->Canvas->MoveTo(5,5);
        Image->Canvas->LineTo(310,5);
        Image->Canvas->LineTo(310,310);
        Image->Canvas->LineTo(5,310);
        Image->Canvas->LineTo(5,5);
        Image->Canvas->Pen->Width=2;
        Image->Canvas->MoveTo(8,8);
        Image->Canvas->LineTo(308,8);
        Image->Canvas->LineTo(308,308);
        Image->Canvas->LineTo(8,308);
        Image->Canvas->LineTo(8,8);
}
//---------------------------------------------------------------------------
```

3. 处理用户连接

在 FormCreate 函数的最后调用 WaitItemClick 函数，此函数也是“等待”菜单项的 OnClick 事件响应函数。这个函数实际上是一个切换程序，如果服务器已处于监听状态，就退出监听状态，如果服务器不在监听状态，就让服务器进入监听状态。当服务器处于监听状态，“等待”菜单项旁边会显示对号标识。函数内容如下：

Source “监听”状态切换和用户连接实现

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::WaitItemClick(TObject *Sender)
{
        WaitItem->Checked = !WaitItem->Checked;
        if (WaitItem->Checked) {
              ClientSocket1->Active = false;
              ServerSocket1->Active = true;
              StatusBar1->Panels->Items[0]->Text = "Waiting...";
```

```
        }
        else{
            if (ServerSocket1->Active)
                ServerSocket1->Active = false;
                StatusBar1->Panels->Items[0]->Text = "";
        }
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::ConnectItemClick(TObject *Sender)
{
        if (ClientSocket1->Active)
            ClientSocket1->Active = false;
        if (InputQuery("我的计算机连接到...", "连接地址:", Server))
        {
            if (Server.Length() > 0)
            {
                ClientSocket1->Host = Server;
                ClientSocket1->Active = true;
            }
        }
}
//---------------------------------------------------------------------------
```

ConnectItemClick 函数为用户单击“连接”菜单项的响应函数。任何一个程序可以作为客户程序向已经处在等待状态的程序提出连接请求，待服务器接受了连接请求后双方才可以进行游戏。客户程序提出请求是通过单击“连接”菜单项或者工具栏的连接按钮完成的。

这个程序用于向服务器端程序发出连接请求。如果客户与服务器已连接，就断开连接，接着在一个询问框内键入新的要连接的服务器的 IP 地址或主机名，然后建立连接。当客户提出连接请求时，在服务器端和客户端都将触发 OnConnect 事件。这之后当服务器接受了客户的连接请求时，将触发 OnAccept 事件。在客户机程序的 OnConnect 事件和服务器的 OnAccept 事件中将操作有关游戏的进一步处理。

12.3.2 实现联机游戏系统

1. 游戏状态的初始化

在客户机程序进行连接时会触发 OnConnect 事件，处理这个事件，完成客户端游戏状态初始化工作。

```
void __fastcall TMainForm::ClientSocket1Connect(TObject *Sender,
```

```
    TCustomWinSocket *Socket)
{
    StatusBar1->Panels->Items[0]->Text = "成功连到：" + Socket->RemoteHost;
    Image1->Picture->LoadFromFile("white.bmp");
    Image2->Picture->LoadFromFile("black.bmp");
    Send[0]='N';strcpy(&Send[1],mName.c_str()); // 构造发送信息
    ClientSocket1->Socket->SendText(AnsiString(Send));
    canPut=false;
    mColor=sWhite;
}
```

该函数作了一些非常重要的工作。首先，确定了游戏双方的执子（在本程序中规定服务器方执黑子先行），并显示出来（在 Image1 和 Image2 中绘制黑子和白子）；其次，客户机程序向服务器端发送了第一条信息，该信息以字母 N 开头，标识信息类型，其后跟随客户机端用户的匿名。发送这条信息的目的是将自己的名字告诉对方。

canPut 是五子棋程序中的一个布尔类型变量，在本函数中将 canPut 变量设为 false，表示等待对方行棋。

mColor 变量用于纪录本机使用的棋子颜色。

当服务器程序接受了客户程序的连接请求，将触发 OnAccept 事件，处理该事件，完成服务器端游戏状态初始化工作。

```
void __fastcall TMainForm::ServerSocket1Accept(TObject *Sender,
    TCustomWinSocket *Socket)
{
    IsServer = true;
    StatusBar1->Panels->Items[0]->Text = "成功连到： " + Socket->RemoteAddress;
    Image1->Picture->LoadFromFile("black.bmp");
    Image2->Picture->LoadFromFile("white.bmp");
    Send[0]='N';strcpy(&Send[1],mName.c_str());
    ServerSocket1->Socket->Connections[0]->SendText(AnsiString(Send));
    canPut=true;
    mColor=sBlack;
}
```

这个函数与客户端 OnConnect 事件处理函数作用相同，它同样构造了一条用于发送自己名字的信息，并将该信息发送出去。同时，IsServer 布尔变量设为 true 表示该程序为服务器端；canPut 变量被设为 true，表示目前我方可以走一步棋。

2. 游戏系统

游戏的进行依赖于 Socket 之间的信息传递。TClientSocket 组件和 TServerSocket 组件在程序中扮演了重要角色，在本程序中传递的信息包括三种类型：一是在连接完成时互报姓名的信息；一是用户的聊天信息，包含了用户的发言内容；最后是行棋信息，包含了用户在棋盘的哪一位置放下了棋子。图 12-5 给出了游戏的进行状态。

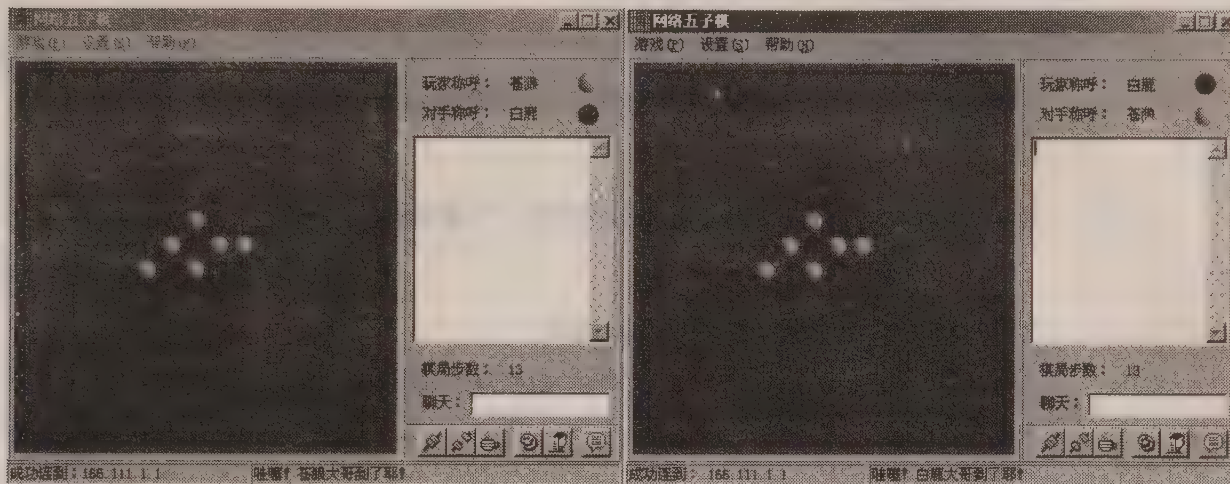


图 12-5 游戏进行中的状态，双方正在交替行棋

对于三类不同的信息，我们定制了一个规则用于信息类型的标识。在程序中，使用字母 N、C、P 作为上述三类信息的信息头，并将信息头附加在要传输的信息上。

游戏中传递信息通过调用 Socket 对象的 SendText 方法（在本程序中，所有发送数据都被格式化成字符串）。而接受信息在 TClientSocket 组件的 OnRead 事件或 TServerSocket 组件的 OnClientRead 事件响应函数中读取并处理。

走棋是通过单击棋盘上的相应位置进行的，所以走棋的处理代码包含在 Image 组件的 OnMouseDown 事件相应函数中。

Source 行棋模块实现

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ImageMouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
      int xx=X/20;  // 算出行棋位置
      int yy=Y/20;
      if (canPut&&Button==mbLeft){ // 是否可以走棋
            if (State[xx][yy]==sNone){  // 该位置是否无棋子
                  State[xx][yy]=mColor;
                  DisplayChess(xx,yy,1);        // 显示棋局
                  Send[0]='P';Send[1]=xx;  // 构造行棋信息
                  Send[2]=yy;Send[3]='\0';
                  if (IsServer)
                        ServerSocket1->Socket->Connections[0]->
                              SendText(AnsiString(Send));
                  else
                        ClientSocket1->Socket->SendText(AnsiString(Send));
                  canPut=false;
                  stepNum++;
```

```
                Label7->Caption=IntToStr(stepNum);
                if (Check(mColor)){ // 是否胜负已分
                    MessageBox(Handle,("恭喜你，"+mName+"，你赢了！").c_str(),
                        "胜出",MB_OK);
                    MessageBox(Handle,"请再接再励！","重新来过",MB_OK);
                    initialize();
                    if (mColor==sBlack)  canPut=true;
                }
            }
        }
}
//---------------------------------------------------------------------------
```

当用户在聊天文本框中输入发言后，单击“发言”按钮可以发送聊天信息，实现聊天功能。

Source 简单聊天功能实现

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ToolButton1Click(TObject *Sender)
{
    Send[0]='C';// 构造聊天信息
    strcpy(&Send[1],Edit1->Text.c_str());
    // 发送信息
    if (IsServer){
        ServerSocket1->Socket->Connections[0]->SendText(AnsiString(Send));
    }
    else{
        ClientSocket1->Socket->SendText(AnsiString(Send));
    }
    // 在本机显示信息
    Memo1->Lines->Add("["+mName+"]");
    Memo1->Lines->Add(Edit1->Text);
    Edit1->Text=EmptyStr;
}
//---------------------------------------------------------------------------
```

本程序的核心部分是服务器端或客户端的信息读取函数。例如，客户端在 TClientSocket 组件的 OnRead 事件响应函数中读取信息。通过信息头判断信息的类型，并做出不同的处理。该函数如下：

Source 读取和分析收到的信息

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ClientSocket1Read(TObject *Sender,
    TCustomWinSocket *Socket)
{
        strcpy(Receive,Socket->ReceiveText().c_str());
        switch (Receive[0]){
        case 'N': // new freid
                yName=AnsiString(&Receive[1]);
                Label4->Caption=yName;
                break;
        case 'C': // chat message
                Memo1->Lines->Add("["+yName+"]");
                Memo1->Lines->Add(AnsiString(&Receive[1]));
                break;
        case 'P': // play chess
                State[Receive[1]][Receive[2]]=sBlack;
                canPut=true;
                DisplayChess(Receive[1],Receive[2],2);
                stepNum++;
                Label7->Caption=IntToStr(stepNum);
                if (Check(sBlack)){
                        MessageBox(Handle,("真抱歉，"+mName+"，你输了！").c_str(),
                                "败北",MB_OK);
                        MessageBox(Handle,"请重新来过！",
                                "重新来过",MB_OK);
                        initialize();
                        canPut=false;
                }
                break;
        }
}
//---------------------------------------------------------------------------
```

该函数意思很明确。对于走棋信息，程序更新了 State 数组，重新显示了棋盘状态，显示了行棋步数，并且进行了胜负判断。当服务器端用户胜出时，程序将显示提示信息并重新开局。

服务器端在 TServerSocket 组件的 OnClientRead 事件的响应函数中读取信息。实现方式与客户端基本相同。

DisplayChess 函数用于在棋盘某点绘制一个棋子。

```
void __fastcall TMainForm::DisplayChess(int xx, int yy,int flag)
{
    if (flag==1)
        Image->Canvas->Draw(xx*20,yy*20,Image1->Picture->Graphic);
    else
        Image->Canvas->Draw(xx*20,yy*20,Image2->Picture->Graphic);
}
```

Check 函数用于判断是否有某一方已经胜出（五子连珠），该函数通过 State 数组对横、竖、左斜和右斜方式的五子相连进行判断。

```
bool __fastcall TMainForm::Check(TState flag)
{
    // 胜负判定函数
    for (int i=0;i<15;i++)
        for (int j=0;j<15;j++){
            if(State[i][j]==flag){
            if (i<11){
                if (State[i][j]==flag&&State[i+1][j]==flag&&
                State[i+2][j]==flag&&State[i+3][j]==flag&&
                State[i+4][j]==flag)
                    return true;
            }
            if (j<11){
                if (State[i][j]==flag&&State[i][j+1]==flag&&
                State[i][j+2]==flag&&State[i][j+3]==flag&&
                State[i][j+4]==flag)
                    return true;
            }
            if (j<11&&i<11){
                if (State[i][j]==flag&&State[i+1][j+1]==flag&&
                State[i+2][j+2]==flag&&State[i+3][j+3]==flag&&
                State[i+4][j+4]==flag)
                    return true;
            }
            if (j<11&&i>=4){
                if (State[i][j]==flag&&State[i-1][j+1]==flag&&
                State[i-2][j+2]==flag&&State[i-3][j+3]==flag&&
                State[i-4][j+4]==flag)
                    return true;
            }
```

```
            }
        }
        return false;
    }
```

12.3.3 简单的辅助功能

网络五子棋程序中提供了更换棋盘背景图案和改变棋盘线颜色的辅助功能。实现改换棋盘线颜色功能用到了 TColorDialog 标准对话框组件，这个组件的 Color 属性返回用户选择的颜色，如图 12-6 所示。

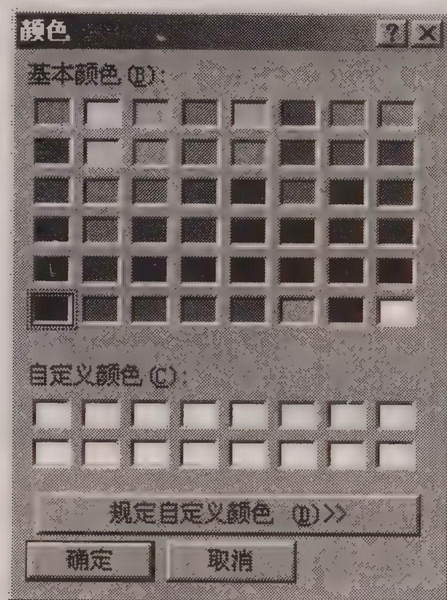


图 12-6　在程序中弹出颜色选择对话框

辅助功能的实现非常简单，此处不再赘述。

至此，具有简单的连网对战功能的网络五子棋程序已基本完成。还有一些细节的问题和处理过程请读者参看程序源码。

12.4　WinSock 编程高级话题

上一节通过创建实例程序“网络五子棋”演示了 C++ Builder 中的 WinSock 编程技术。程序实现了服务器端和客户端的实时通信，但在服务器程序和客户机程序之间传输的仅仅是简单的文本信息。实际上，使用 Socket 对象的 SendText 方法传送的字符串大小是有限制的，而另一种简单的传输方式，在两端使用 SendBuf 和 RecieveBuf 方法发送和接收数据也有同样的问题。所以，一般在服务器和客户间传输大量数据时，流式传输是最为合适和方便的。

12.4.1 流类数据传输

Socket 连接基于两种方式：非阻塞方式连接和阻塞方式连接。Socket 连接上的读写操作异步发生，读写操作不影响应用程序中其他代码的执行，这种方式的连接称为非阻塞方式连

接。Socket 以同步方式读写数据，在读或写操作完成之前，应用程序处于等待状态，这种方式的连接称为阻塞方式连接。

非阻塞方式是缺省的连接方式，上一节创建的网络五子棋程序就采用此方式。在这种情况下，两个 Socket 组件的 ClientType 属性和 ServerType 属性为缺省值，ctNonBlocking。非阻塞方式连接下，服务器端和客户端的 Socket 组件将自动为每一个连接创建一个线程，以保证读写操作不影响其他代码的执行。

在非阻塞方式下，可以使用实际 Socket 对象的 SendStream 方法传输一个流类对象，例如一个文件、声音或者图像数据等等。使用 SendStream 发送流解决了传输数据大小受限制的问题。发送端的数据流可以一步到位。然而，在接受端的 Socket 对象并没有相应的 RecieveStream 方法将发送过来的流全盘接受。解决的方法是在接受数据端使用 RecieveBuf 方法分段接受数据。

RecieveBuf 对象方法有一个有用的特性，它可以返回实际读取的字节数，在程序中我们可以在一个 while 循环中反复使用 RecieveBuf 函数从接受的流中读取数据，当 RecieveBuf 函数读取的字节数小于 0 时，退出循环并结束读取。程序示例如下：

```
char Buffer[1000];
TMemoryStream *myStream;
do {
    n=Socket->ReceiveBuf(Buffer,sizeof(Buffer));
    if (n<=0)
        break;
    else
        myStream->Write(Buffer,n);
        Sleep(200); // 延迟 200 毫秒
} while(true);
```

程序中通过 Sleep 函数造成读数据过程的 200 毫秒的硬延迟，这是非常必要的。因为在接收端的从流中读取数据的速度较快，而发送端使用 SendStream 传输流的速度较慢，数据到达速度不可能跟上读取数据的速度。如果没有时间延迟，则会造成接收端提前无数据可读，而导致循环中止，数据丢失。

这种解决方案显然不够完美。其一，设定硬延迟的时间不好掌握；其二，硬延迟时间会造成程序效率的大大降低。但在非阻塞的连接方式下也只能如此。在本章的随后部分，将会看到两种更好的流式数据传输的解决方案。

12.4.2 利用 WinSock 定制协议

前面曾经谈到，Internet 上常见的标准协议如 HTTP、FTP 等，其实都是建立在 TCP/IP 协议基础之上的。在 WinSock 编程中，除非仅仅是完成简单的传输任务，否则也应该在服务器程序和客户机程序之间建立自己的一套通信规则，也就是所谓的协议。

协议是客户机与服务器之间用于确定通信流程的一系列规则。具体来说，这一套规则被用来确定客户机可以发往服务器的请求（Request），以及服务器如何响应各种不同的请求。所以，定制协议的关键是定义不同类型的请求，并具体实现这些请求的回应。

下面将举出一个实例程序 Protocol，该程序定义了简单的文件传输协议（有点类似 FTP）。这个程序中仅定义并处理两种请求：获得文件列表请求（类似 FTP 协议中的 nList）和下载文件请求（类似 FTP 协议中的 Download）。

图 12-7 给出了 Protocol 范例的客户机程序。

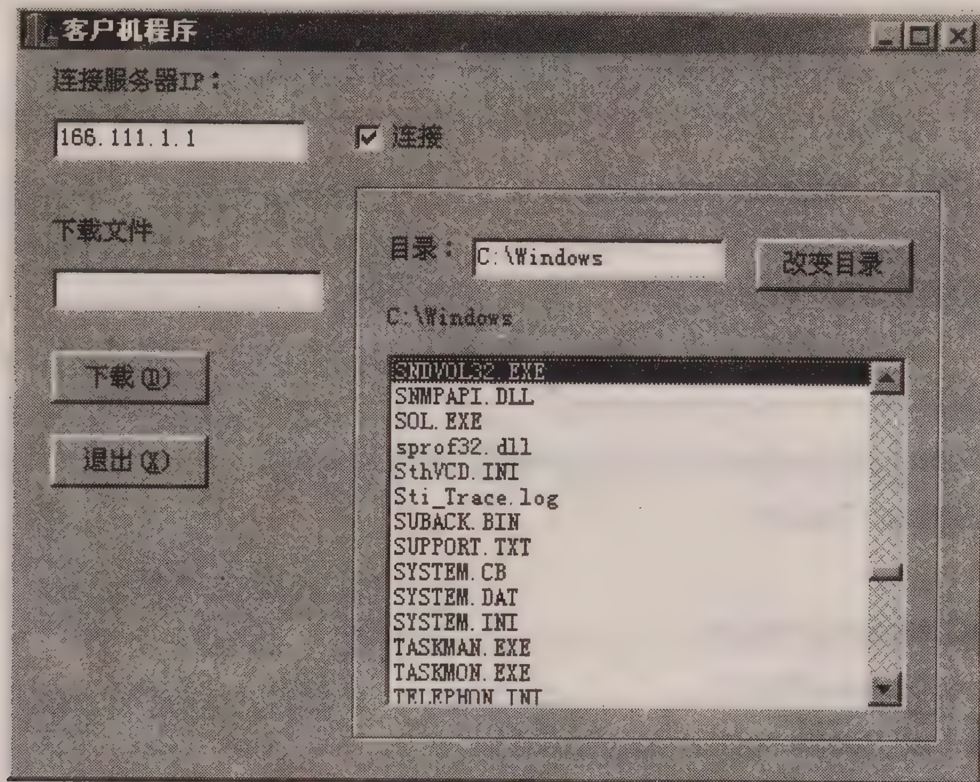


图 12-7 Protocol 范例的客户机程序

当客户端程序与服务器端程序正常连接后，客户端用户可以向服务器提出请求。通过“下载”按钮和“改变目录”按钮可以发出两种不同类型的请求。程序中对这两种请求的信息分别增加了“LIST_”和“FILE_”五个字符的信息头。如下所示：

Source 请求信息的构造及发送

```
//---------------------------------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
        AnsiString Msg;
        Msg="LIST_"+Edit2->Text; // 构造请求信息
        ClientSocket1->Socket->SendText(Msg);
}

//---------------------------------------------------------------------------

void __fastcall TForm1::Button2Click(TObject *Sender)
{
        if (SaveDialog1->Execute()){
                FileName=SaveDialog1->FileName;
```

```
        AnsiString Msg;
        Msg="FILE_"+Edit3->Text; // 构造请求信息
        ClientSocket1->Socket->SendText(Msg);
    }
}
//---------------------------------------------------------------------------
```

在客户程序发出这些请求信息后，服务器程序的 ClientRead 函数将读取并处理这些请求。ServerSocket1ClientRead 函数如下：

Source 请求信息的分析和响应

```
//---------------------------------------------------------------------------
void __fastcall TForm1::ServerSocket1ClientRead(TObject *Sender,
    TCustomWinSocket *Socket)
{
    AnsiString rCommand,rFile,Msg;
    Msg=Socket->ReceiveText();
    rCommand=Msg.SubString(1,5); // 分解信息头
    rFile=Msg.SubString(6,Msg.Length()-5); // 分解文件名
    if (rCommand=="LIST_"){
        if(DirectoryExists(rFile)){
            Socket->SendText(rCommand);
            FileListBox1->Directory=rFile;
            Socket->SendText(FileListBox1->Items->Text);
        }
    }
    else if (rCommand=="FILE_"){
        if (FileExists(rFile)){
            Socket->SendText(rCommand);
            TFileStream *fileStream=
                new TFileStream(rFile,fmOpenRead|fmShareDenyWrite);
            Socket->SendStream(fileStream);
            fileStream->Free();
        }
    }
}
//---------------------------------------------------------------------------
```

在这段程序中，首先从接收信息中解析出信息头和实际信息（文件名或目录名）。然后，程序根据信息头的不同，向客户程序发回了不同的信息。

处于设计期间的服务器端程序界面如图 12-8 所示。



图 12-8　设计期间的服务器端程序界面

为了传输目录列表，在服务器程序中加入一个 TFileListBox 组件，并将其 Visible 属性设为 false。通过该组件获得指定目录下的文件和子目录的列表，并将列表信息发回。

对于传输文件，在程序中建立一个临时文件流。并使用 SendStream 对象方法发送这个文件流。在发送流式数据之前，服务程序首先会发回信息头作为响应。

在接收了服务器端发送的信息头之后，客户端程序会设置一个状态变量，并等待服务器端程序发送实际数据。设置状态变量是定制协议编程中使用的典型技术。程序中定义了 TClientStatus 枚举类型：

```
enum TClientStatus {csWait,csList,csFile};
```

同时程序定义了 TClientStatus 类型变量 ClientStatus，在客户端的 ClientSocket1Read 函数中使用了该变量，并进一步接收服务器端送来的数据。

```
void __fastcall TForm1::ClientSocket1Read(TObject *Sender,
    TCustomWinSocket *Socket)
{
    int n;
    switch (ClientStatus){
    case csWait:
        Header=Socket->ReceiveText();
        if (Header=="LIST_")
            ClientStatus=csList;
        else if (Header=="FILE_")
            ClientStatus=csFile;
        break;
    case csList:
        ListBox1->Items->Text=Socket->ReceiveText();
        Label3->Caption=Edit2->Text;
        ClientStatus=csWait;
        break;
    case csFile:
        tmpStream=new TMemoryStream();
        Screen->Cursor=crHourGlass;
        do {
            n=Socket->ReceiveBuf(Buf,sizeof(Buf));
            if (n<=0)
```

```
                break;
            else
                tmpStream->Write(Buf,n);
            Sleep(200);
        }while(true);
        tmpStream->SaveToFile(FileName);
        delete tmpStream;
        Screen->Cursor=crDefault;
        ClientStatus=csWait;
        MessageBox(Handle,"下载完毕!","WinSock 编程示例",MB_OK);
    }
}
```

当客户端处在 csWait 状态时，在该函数中将接收信息头，并设定客户端状态，等待进一步接收数据。当客户端处在 csList 状态时，程序将接收文件列表，并将该文件列表显示在一个列表框中；当客户端处在 csFile 状态时，程序将接收服务器端传来的文件流，并且保存文件。

12.4.3 在阻塞状态下传输数据

当服务器程序中的 TServerSocket 组件的 ServerType 属性设为 stThreadBlocking，并且客户程序中的 TClientSocket 组件的 ServerType 属性设为 ctBlocking 时，客户程序和服务器程序将使用阻塞连接。在阻塞连接的情况下，没有如 OnRead 或 OnWrite 等异步事件，Socket 必须主动去读或写数据，在读或写操作完成之前，应用程序处于等待状态，这样就不可避免地会影响整个应用程序的性能。

所以当使用阻塞式连接时，必须在服务器程序上使用一个线程，而在客户机程序中一般也要创建一个相应的线程。在服务器端，TServerSocket 组件会为每一个阻塞方式的连接自动分配一个新的线程。服务器端的 Socket 用 TServerClientThread 对象来操纵每一个线程，当线程开始执行时，它就检查位于另一端的客户是否正在试图进行写。如是，则触发 OnClientRead 事件，否则，触发 OnClientWrite 事件让服务器去写。

在阻塞状态下，可以使用 TWinSocketStream 类进行实际的流类读写操作。使用线程和 TWinSocketStream 类组合进行流类数据传输是一个较为常用的做法。但一个问题是，客户端如何得知对方已准备好读或写呢？这就要用到 TWinSocketStream 对象的 WaitForData 方法，以保证服务器和客户的同步。同时，还可以通过 TimeOut 属性设定超时值，这种超时保护机制可以在传输或连接出现问题时，避免程序无休止地等待下去。

12.5 使用 TPowerSock 组件类

在第 10 章，我们曾经使用了一组实现协议的组件，如 NMHTTP，NMFTP，NMPOP3

等，并且利用这些组件创建了简单的客户端程序。实际上，这一组组件被称为 FastNet Tools 组件，这个组件集合由 NetMasters 公司提供，它代替了 C++ Builder 3 中使用的由 NetManage 公司提供的一组 ActiveX 控件。

这一组 VCL 组件的共同父类是一个被称为 TPowerSock 的组件。该组件是 C++ Builder 本身的 TCleintSocket 和 TServerSocket 组件的一个非常强劲的替代品。事实上，通过 TPowerSock 组件和 TNMGeneralServer 组件完全可以替代 C++ Builder 自己的 Socket 组件，并且做得更好。

12.5.1 TPowerSock 组件

TPowerSock 组件一般用来作为一个创建新协议的基类，包括用户定制协议。从 TPowerSock 组件中可以继承 WinSock 编程所需的基本支持和特殊支持，从 TNMGeneralServer 组件可以继承多线程服务器所需的支持。

TPowerSock 组件和 TNMGeneralServer 组件也可以直接使用，如同 TClientSocket 组件和 TServerSocket 组件一样。但较之 TClientSocket 组件和 TServerSocket 组件，TPowerSock 组件有更为强劲的功能。例如：

- TPowerSock 组件提供 BytesRecvd、BytesSent、BytesTotal 属性，可以直接获得数据的传输进度。
- TPowerSock 组件提供 Proxy、ProxyPort 属性，对通过代理服务器连接提供直接支持。
- 除了 SendBuffer、SendStream 对象方法之外，TPowerSock 组件提供了 SendFile 方法，对文件传输提供了直接支持。
- 读取传来数据的方法包括 CaptureFile、CaptureStream、CaptureString。比使用 C++ Builder 本身的 WinSock 组件更为好用。
- TPowerSock 组件含有一个 TimeOut 属性，支持超时保护机制。

TPowerSock 组件的使用方式与使用 TClientSocket 组件和 TServerSocket 组件基本相同。它提供 Listen 方法用于进入监听状态，Connect 方法用于向服务器端程序提出连接请求。例如，一段典型的连接服务器代码如下：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Powersock1->Host = "166.111.1.1";
    Powersock1->Port = 100;
    Powersock1->ReportLevel = 1;
    Powersock1->TimeOut = 5000;
    Powersock1->Connect();
}
```

与 TClientSocket 组件和 TServerSocket 组件一致，TPowerSock 组件也提供了 OnRead 事件，在这个事件的响应函数中可以进行传来数据的读取。

12.5.2 TNMStrm 和 TNMStrmServ 组件

除了 TPowerSock、TNMGeneralServer 组件以及各种高级协议组件之外，FastNet Tools 组件集合还包含了一些备用 Socket 组件。其中，NMMsg 组件和 NMMsgServ 组件用于在连接上发送和接收文本信息，NMStrm 组件和 NMStrmServ 组件用于发送和接收流式数据。使用 NMStrm 组件和 NMStrmServ 组件发送流式数据变得非常简单。图 12-9 给出了 TNMstrm 组件示例程序。

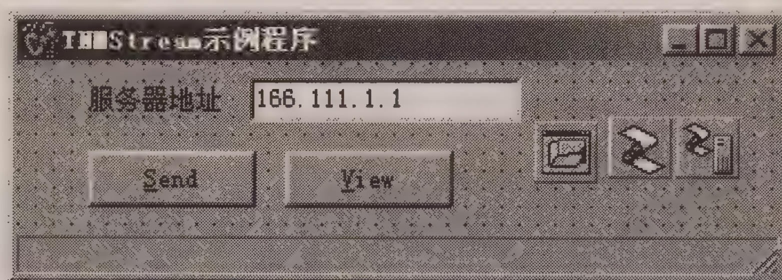


图 12-9 设计期间的 TNMStrm 组件示例程序

在此通过创建一个传输位图图像的程序，演示 TNMStrm 和 TNMStrmServ 组件的使用。TNMStrm 和 TNMStrmServ 组件继承于 TPowerSocket 组件，它的关键方法名为 PostIt，用于直接传输流式数据。

Send 按钮用于发送位图数据流，首先通过 TNMStrm 组件的 Host 属性设置主机 IP，然后调用 PostIt 对象方法一步完成与服务器的连接和流式传输。实现函数如下：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    if (OpenPictureDialog1->Execute())
    {
        TFileStream *tmpStream;
        tmpStream = new TFileStream(OpenPictureDialog1->FileName,fmOpenRead);
        try {
            NMStrm1->Host = Edit2->Text;
            NMStrm1->PostIt(tmpStream);
        }
        catch(...)
        {
        }
        tmpStream->Free();
    }
}
```

图 12-10 显示了在范例程序中流式传输位图文件的过程。

响应 TNMStrmServ 组件的 OnMsg 事件，接收流式数据。TNMStrm 组件的传输机制使

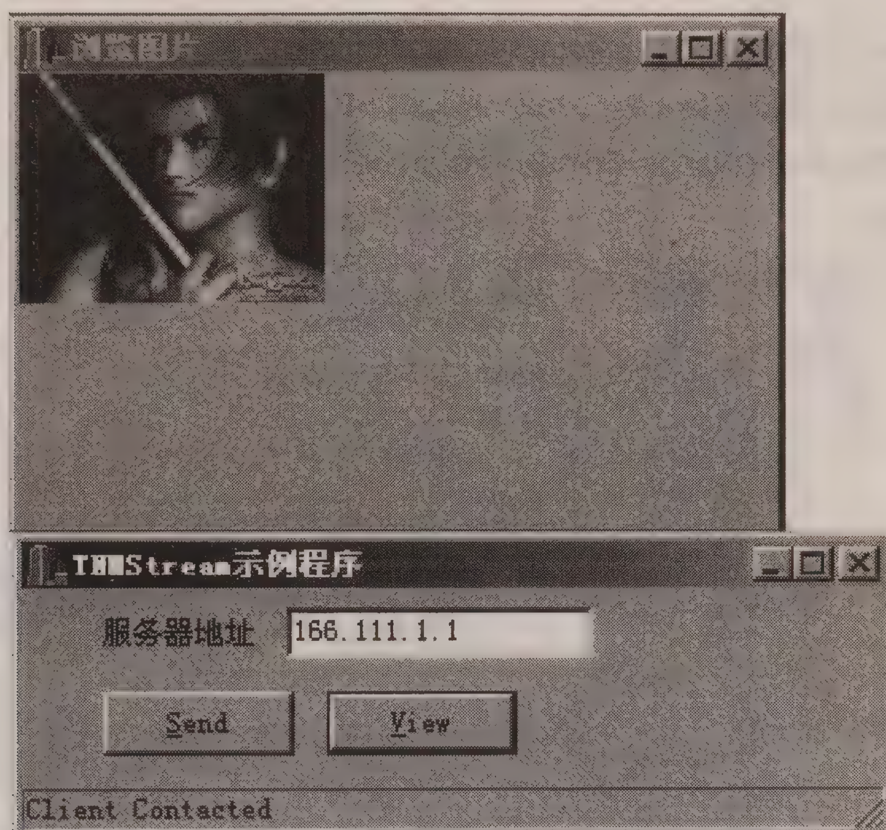


图 12-10　在范例程序中流式传输位图文件

得我们不必再进行硬延迟，传来的流式数据直接被接收，可以通过 OnMsg 事件处理函数的 strm 参数访问。此时可以将该流存储在临时文件流中，该临时文件流对应硬盘上的一个临时文件。实现函数如下：

```
void __fastcall TForm1::NMStrmServ1MSG(TComponent *Sender,
    const AnsiString sFrom, TStream *strm)
{
    if (FileExists(".\\tmp.bmp"))
        DeleteFile(".\\tmp.bmp");
    TFileStream *tmpStream;
    tmpStream = new TFileStream(".\\tmp.bmp", fmCreate);
    try
    {
        tmpStream->CopyFrom(strm, strm->Size);
    }
    catch(...)
    {
    }
    tmpStream->Free();
}
```

当用户按下 View 按钮，程序将在另一个窗体中显示接收到的位图。注意调整窗体的 HorzScrollBar 和 VertScrollBar 的 Range 属性，生成窗体滚动条并使之适合图像大小。具体代码如下：

```
void __fastcall TForm1::Button2Click(TObject *Sender)
```

```
{
    Form2->Image1->Picture->LoadFromFile(".\\tmp.bmp");
    Form2->HorzScrollBar->Range=Form2->Image1->Picture->Width;
    Form2->VertScrollBar->Range=Form2->Image1->Picture->Height;
    Form2->Show();
}
```

第13章

图像时钟组件

——创建VCL组件

C++ Builder 组件和组件包技术
为已有组件增强功能
创建图像时钟组件
C++ Builder 内部消息
自定义事件
创建非可视组件

本章导读

创建组件在 C++ Builder 中非常容易，这可能是 C++ Builder 最具特色的地方。同时，所创建的 VCL 组件具备很多明显优势，VCL 的组件比起流行的 ActiveX 控件来说，规模小得多，但功能却强大得多且效率很高。本章讲述使用 C++ Builder 进行 VCL 组件开发以及组件编程的各种技术。创建组件实例是一个功能强大的图像时钟组件 TCoolClock。该组件实现了一个大小、样式和颜色可调的实时时钟。同时本章还包括其他一些组件范例，它们分别负责说明 C++ Builder 中创建组件技术的某个方面。

本章内容包括：

- C++ Builder 组件和组件包技术的讨论，基于抽象组件类创建简单组件实例。
- 基于实际组件创建功能增强的组件。
- 基于基本控制类创建组件，具体实现图像时钟组件 TCoolClock。并讨论利用消息映射使用 C++ Builder 内部消息，增加 TPersistent 属性，增加枚举类型属性，为组件创建新定义事件等问题。
- 创建非可视化组件。

13.1 C++ Builder 组件和组件包

13.1.1 扩展 VCL 组件

从狭义的角度讲，组件是在组件选项板上选择的，并在窗体设计窗口和代码窗口中操作的元素。从广义角度讲，组件是任何代码中的定义类，是能插入 C++ Builder 开发环境的任何元素。组件可能具有程序的各种复杂性，但提供了简单易用的使用接口，并隐藏了所有的实现细节。这正是应用组件编程渐渐成为开发应用的主要方法的原因所在。

在 C++ Builder 中编写组件的本质是对 VCL 可视化组件库的扩展，因为所有的 C++ Builder 组件创建都必须继承于一个 VCL 的基类（当然，也可以是用户所编写的 VCL 组件类）。但这并没有限制 C++ Builder 中创建组件的性能，相反，完全面向对象（OPP）的继承创建方式使得开发人员更容易做出功能强大的自定义组件。

利用 C++ Builder 创建组件是简单的，如果读者曾经开发过 ActiveX 控件的话，一定会对 C++ Builder 优越的组件创建机制感到惊叹。在 C++ Builder 中，只要能融入组件框架，组件基本上就是在应用程序中为实现相同功能而所编写代码的一切，基本上不用做其他与组件实际性能无关的工作。创建组件唯一不便的是，它是一个完全非可视化的过程。

派生一个新组件的工作可以从已有组件或者抽象组件类开始。在 VCL 层次结构中包含很多这样的抽象组件类，这样的设定为扩展 VCL 组件性能或开发新组件提供了非常便利的条件。

13.1.2 创建组件的原则

在 C++ Builder 中创建组件与编写应用程序有明显的不同。这表现在以下两点：

- 编写组件的过程是非可视化的。
- 组件编写完全以代码的形式进行，即非可视化。因为 C++ Builder 可视化设计的实现基于已完成的组件，而建立这些组件需要用 C++代码编写。

虽然创建组件是非可视化的，但这并不影响在编写代码时使用 VCL 可视化组件。例如，完全可以在组件编写中加入一个 TTimer 定时器对象或 TImage 图像对象，并利用这些组件的功能达到自定义组件的某些需要。

1. 编写组件需要更深入的 OPP 知识

除了非可视化编程之外，建立组件和使用组件存在明显区别：当建立新组件时，需要从已存在组件或类中继承产生一个新对象类型，并增加新的属性、方法和事件；另一方面，在开发人员使用组件建立 C++ Builder 应用时，则是在设计阶段通过改变组件属性和描述响应事件的方法来定制它们的行为，并通过程序控制组件对象在运行时的行为。

编写组件要求开发者必须有更深入的 OPP 知识。这是因为 C++ Builder 组件创建规则完全遵循 OPP（面向对象编程）方法。类的继承、多态性、重载等概念经常会用到。

在 C++ Builder 中编写组件一般应遵循下面的原则：

- 尽量使自己创建的组件和 C++ Builder 已有组件风格相同。这要求对 VCL 层次结构和已有组件有深入的了解。
- 创建组件应遵循 Borland 命名规则。对于组件来说，属性、事件和方法都有各自的命名规则，遵循这些规则可以使其他人更容易使用我们开发的组件，并易于进一步扩展它们。
- 保持组件的易用性和简单化。对于我们创建的组件来说，无论具体的实现代码是多么复杂，提供给用户的接口应与 C++ Builder 已有组件一样易于使用。同时，应该尽量利用相似的属性、对象方法和事件名称。
- 使用异常保证组件的完善性。在创建组件中特别不应该忽略异常的使用，当出现错误时，组件应该引起一个异常。
- 为组件创建合适的位图。完成组件后，要为该组件创建一个有良好提示作用的位图，这个位图会显示在组件选项板上。一般不要使用 C++ Builder 自动提供的位图。
- 为组件创建帮助，创建帮助应该尽量保持和 C++ Builder 中已有组件帮助的风格一致。最后可以用 C++ Builder 提供的 OpenHelp 工具将帮助加入到 C++ Builder 帮助中。

13.1.3 组件包

C++ Builder 使用组件包为组件应用提供支持。每个组件包实质上都是一种扩展名为 BPL (Borland Package Library）的 DLL。

组件包分为两种：C++ Builder IDE 使用的设计期间组件包和应用程序使用的运行期间组件包。这两种组件包分别对应 Design-Only 标志和 Read-Only 标志。

每一个组件包可以包含多个不同组件。一个组件包的项目文件（BPK 文件，Borland Package）包含了这个组件包中的组件信息和编译信息。

创建完成的组件必须通过组件包安装到开发环境中。我们可以新建一个组件包，或将组件加入一个已有的组件包。C++ Builder 提供一个缺省的自制组件的组件包 User Components。

13.1.4 创建一个简单的组件

以上已经了解了创建 C++ Builder 组件的基本概念，应该明确以下三点：

- 创建组件继承于一个现有的 VCL 组件。
- 创建组件的主要任务是为组件增加新的属性、方法和事件。
- 使用组件包使创建组件与 C++ Builder IDE 紧密结合。

下面创建一个可视组件 TClockEdit，这个组件实现了一个在文本框中显示的数字时钟。此组件继承于 VCL 中的 TCustomEdit 组件，TCustomEdit 组件是 TEdit 组件的抽象类，

TCustomEdit 组件与 TEdit 组件有相同的功能，只是几个属性没有公布（Publish）出来。使用 TCustomEdit 组件可以在创建组件版本中隐藏 TEdit 组件的一些标准属性。

在 C++ Builder 中创建一个新组件可以选择 Component 菜单下的 New Component 命令启动 Component Wizard，或者选择【File】菜单下的【New】命令打开 New Items 对话框，并在 New 页中选择 Component，也可以启动 Component Wizard。在 Component Wizard 中需要填写下列信息，如图 13-1 所示。

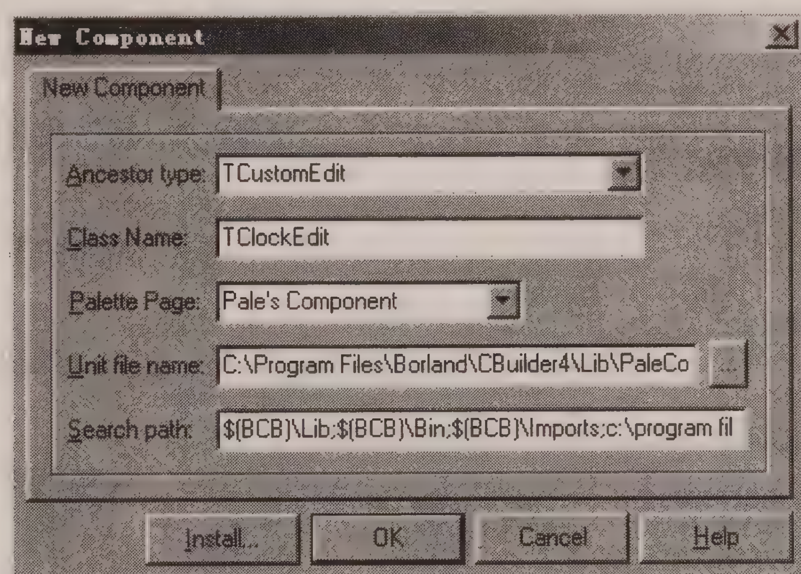


图 13-1 在 C++ Builder 中创建新的组件

- 父类型的名称。在本例中，选择 TCustomEdit。
- 建立组件的名称。在本例中，填写 TClockEdit。
- 组件在 Component Palette 的页面。在本例中，我们创建了一个新页面：Pale's Component。
- 组件的 CPP 文件名。
- 当前搜索路径。一般不用修改。

按下 OK 按钮，组件向导将自动生成新建组件的框架。如果单击 Install 按钮，则会直接将组件安装到组件包中。建议使用 OK 创建新组件，这样可以在组件调试成功后再将组件加入组件包文件中。

此后做的工作是在组件的头文件中更改组件类的定义，增加新的属性和方法，对于本例来说，还应该在__published 域中重新声明父类 TCustomEdit 中的一些属性使其公布出来。

以下是 TClockEdit 组件类的完整声明：

Source 新建组件 TClockEdit 的声明

```
class PACKAGE TClockEdit : public TCustomEdit
{
private:
    TTimer *FTimer;
    bool _fastcall GetActive(void);
    void _fastcall SetActive(bool Value);
```

```
protected:
        void _fastcall UpdateIt(TObject* Sender);
        __fastcall ~TClockEdit(void);
public:
        __fastcall TClockEdit(TComponent* Owner);
__published:
        __property Align;
        __property Color;
        __property Font;
        __property PopupMenu;
        __property ShowHint;
        __property Visible;
        __property OnChange;
        __property OnClick ;
        __property OnDblClick ;
        __property OnMouseDown ;
        __property OnMouseMove ;
        __property OnMouseUp ;
        __property bool Active={read=GetActive, write=SetActive, default=true};
};
```

这个简单的组件实例不仅说明在 C++ Builder 中如何创建组件，同时它也说明了如何在创建组件中使用其他 VCL 组件，以及如何为组件增加一个属性。

在组件的 private 域声明了一个 TTimer 类型的内部对象。组件中内部变量或对象命名以 F（Field）字开头，VCL 类库中所有类都遵循这个命名规则。

为了创建定时器对象，必须在类的构造函数中将其实例化。下面是 TClockEdit 组件类单元文件中的部分内容：

Source TClockEdit 组件的实现部分

```
static inline void ValidCtrCheck(TClockEdit *)
{
        new TClockEdit(NULL);
}
__fastcall TClockEdit::TClockEdit(TComponent* Owner)
        : TCustomEdit(Owner)
{
        Text=TimeToStr(Time());
        ReadOnly=true;
        AutoSelect=true;
        FTimer=new TTimer(this);    // 创建定时器对象
```

```
        FTimer->Interval=1000;
        FTimer->OnTimer=UpdateIt;
        FTimer->Enabled=true;
}
__fastcall TClockEdit::~TClockEdit(void)
{
        delete FTimer;
}
bool _fastcall TClockEdit::GetActive(void)
{
        return FTimer->Enabled;
}
void _fastcall TClockEdit::SetActive(bool Value)
{
        FTimer->Enabled=Value;
}
void _fastcall TClockEdit::UpdateIt(TObject* Sender)
{
        Text=TimeToStr(Time());
}
namespace Clockedit
{
        // 注册组件代码，由 C++ Builder 自动添加，无需改动
        void __fastcall PACKAGE Register()
        {
                TComponentClass classes[1] = {__classid(TClockEdit)};
                RegisterComponents("Pale's Component", classes, 0);
        }
}
```

在类的构造函数中做的重要工作是为 TTimer 对象的 OnTimer 事件提供了处理函数，这使得定时器对象真正能够起到作用：每隔 1 秒，将当前的时间写到文本框中。这里调用了 Time 全局函数，该函数返回当前系统时间。图 13-2 显示出了 TClockEdit 组件的应用。

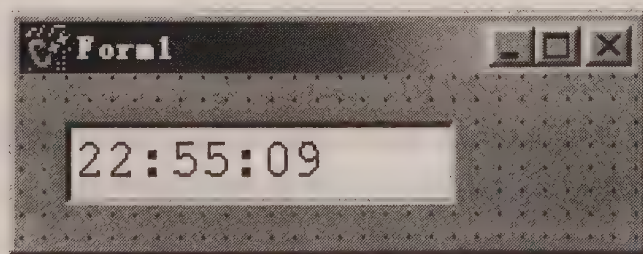


图 13-2 测试 TClockEdit 组件

显然不能让用户直接控制组件的 TTimer 对象，所以将 TTimer 对象声明在私有部分。但

另一方面，TTimer 对象的 Enabled 属性理应交给用户控制，使组件使用者决定时钟运行和停止。为此给 TClockEdit 组件添加一个新属性 Active。

在自定义组件中，属性的一般声明方式为：

__property 类型 属性名 = {read=函数名或变量名, write=函数名或变量名, default=缺省值};

其中 read，write 和 default 三部分可以不全，除了使用“default = ”方式声明缺省值之外，还可以使用 nodefault 声明该属性无缺省值。

自建的新属性如图 13-3 所示。

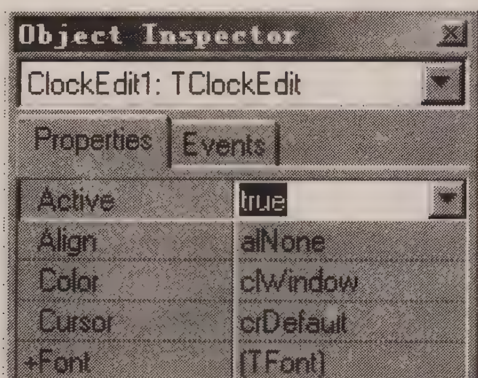


图 13-3 TClockEdit 组件的属性页，包含了自建的 Active 属性

在__published 域还重新声明了一些父类 TCustomEdit 为公布的属性和事件，这些公布的属性和事件将自动启用。而没有重新声明的属性或事件，将会继续隐藏下去。本组件没有公布组件的 Text 属性，因为不希望用户改动编辑框的文本，这正是选择 TCustomEdit 作为父类而不选 TEdit 的原因。

13.2 组件编程

通过上一节的例子，应该大致明白 C++ Builder 中创建组件的方法。本节将对如何在 C++ Builder 中创建 VCL 组件作系统阐述。

13.2.1 创建组件的起点

一般地，建立新组件意味着从已有类型中继承得到新组件对象类。一般来说，建立新组件有以下几种方案：

- 从已有组件的抽象类派生组件。
- 直接从已有组件派生组件。
- 从 TWinControl 类或 TCustomControl 类派生具有窗口特性的可视化组件。
- 从 TGraphicControl 类派生具有图形特性的可视化组件。
- 从 TComponent 类派生非可视化组件。

当然，也可以继承其他广义组件对象，但无法在窗体设计窗口中操作它们。C++ Builder 包括许多这种对象，如 TCanvas、TBitmap、TList 等。

1. 从已有组件的抽象类派生组件

VCL 可视化组件库的良好结构在于它为很多组件提供了抽象类，同一个抽象类可以派生出不同特性的现实组件。最明显的例子是 TEdit、TMemo、TRichEdit 和 TMaskEdit 组件都是从 TCustomEdit 中派生出来的。现实组件的抽象类都以 TCustomXxxx 命名，可以从抽象类出发创建许多不同的现实组件。例如，开发人员也许想建立 TTreeView 的特殊类型，这种组件不具有标准 TTreeView 的某些属性，当然不能将继承自父类的属性从父类中移去，因此开发者必须从比 TTreeView 更高层次的组件继承，例如 TCustomTreeView。事实上，TCustomTreeView 组件实现了 TTreeView 的所有属性并将它们全部保护(protected)起来。当组件从诸如 TTreeView 的抽象类中继承时，开发者可以控制公布那些别人想获得的属性而让其他的属性及事件继续隐藏。

上一节创建的 TClockEdit 组件就是一个这样的实例。

2. 直接从已有组件派生组件

直接从已有组件派生的组件可以继承父类型的所有属性，这样派生新组件往往有两种目的：

- 定制原有组件。通过给原有组件的某些属性赋值，创建一个有特定用途的子类。例如，可以创建一个继承于 TComboBox 的组合框组件，将其 Items 属性赋为当前支持的所有字体，就完成了一个字体选择组合框的组件。
- 创建原有组件的增强类型。直接向已有组件加入新的属性、对象方法和事件，使原有组件更加强大。例如，可以为 TRichEdit 组件增加 CurrentLine 和 CurrentColumn 属性，用于获取或返回当前光标所在的行列。

3. 从 TWinControl 类或 TCustomControl 类派生组件

除了上述两种基于已有组件创建新组件方式，我们往往需要从底层创建组件。从底层创建新组件有几种常用的基类选择，包括从 TWinControl 类或 TCustomControl 类、TGraphicControl 类、TComponent 类继承创建组件。

以 TWinControl 类或 TCustomControl 类为起点派生的组件的关键特征是它们具有窗口过程，从而也就具有窗口句柄，句柄保存在属性 Handle 中，这类组件：

- 能够接收发给窗口过程的 Windows 消息，如 WM_TIMER。
- 能接受输入焦点。
- 能将句柄传送给 Windows API 函数。

4. 从 TGraphicControl 类派生组件

从 TGraphicControl 类派生类似于从 TCustomControl 类派生，但它们没有窗口句柄，因此占有较少系统资源。这类组件最大的限制是它们不能接收输入焦点。然而，TGraphicControl 类提供作图的 Canvas 并且可以处理 WM_PAINT 消息，并且它拥有实的 Paint 方法，在创建时需要重载这个方法。

如果组件不需要接受输入焦点并且不用响应特定的消息，可以考虑从 TGraphicControl 类派生，这样能够节省系统资源，从而使创建的组件更有效率。

5. 从 TComponent 类派生非可视化组件

抽象对象类型 TComponent 是所有组件（包括可视组件和非可视组件）的公共基类，同时是非可视组件的直接父类。从 TComponent 类直接继承所创建的组件就是非可视化组件。TComponent 定义了组件在 FormDesigner 中所需的基本的属性和方法，因此，从 TComponent 继承来的任何组件都具备设计能力。

非可视组件的作用不容忽视，非可视组件可以产生特殊作用。例如各种对话框组件就是非可视组件，像 TImageList，TMenu，TPopMenu 这样重要的组件也是非可视组件。

13.2.2 链接图像组件

本小节将创建直接从已有组件派生组件的范例——链接图像组件 TLinkImage。该组件是在已有组件 TImage 组件之上新创建的，它实现网页中常用的图像链接的效果：当鼠标移动到 TLinkImage 之上，图像会切换为另一个，而当鼠标离开图像时，图像恢复为原有图像。

这个组件增强了 TImage 的功能，同时该组件范例的创建传递给我们一些有益的信息：第一，在组件中利用消息映射处理消息；第二，使用和处理 C++ Builder 内部消息。

以下是 TLinkImage 组件的声明部分。

Source TLinkImage 组件的完整声明

```
class PACKAGE TLinkImage : public TImage
{
private:
    TPicture *FHoverPic;
    TPicture *FLinkPic;
    void __fastcall SetHoverPic(TPicture *Value);
    void __fastcall SetPicture(TPicture *Value);
protected:
    void __fastcall MouseEnter(Messages::TMessage &Message);
    void __fastcall MouseLeave(Messages::TMessage &Message);
public:
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(CM_MOUSEENTER,TMessage,MouseEnter)
    MESSAGE_HANDLER(CM_MOUSELEAVE,TMessage,MouseLeave)
END_MESSAGE_MAP(TImage)
    __fastcall TLinkImage(TComponent* Owner);
    __fastcall ~TLinkImage(void);
__published:
    __property Graphics::TPicture *LinkPic = {read=FLinkPic,write=SetPicture};
```

```
    __property Graphics::TPicture *HoverPic = {read=FHoverPic,write=SetHoverPic};
    __property Picture= {read=FPicture,write=SetPicture};
};
```

该组件继承于 TImage 组件，它定义了另两个 TPicture 类型的属性 LinkPic 和 HoverPic，将会看到 C++ Builder 自动在 Object Inspector 中提供与 TPicture 类型相适应的属性编辑器（支持图像载入和预览的 Pictrue Eidtor 对话框）。

同时这个组件重新声明了 Image 类的 Picture 属性，并重载该属性的 write 方法，这将改变 Picture 属性的性质。

TLinkImage 范例组件的重点演示了组件编程中使用消息映射的技术。TLinkImage 组件需要在用户鼠标移动到图像上和移出图像时做出响应并进行处理，C++ Builder 的内部消息 CM_MOUSEENTER 和 CM_MOUSELEAVE 正好可以完成这样的编程需要。在 TLinkImage 组件的声明中我们加入了对内部消息 CM_MOUSEENTER 和 CM_MOUSELEAVE 的映射处理，可以看到，组件创建中的消息映射和通常编程中的消息映射并无区别。当然，在自定义组件中响应标准 Windows 消息也没有问题。

CM_MOUSEENTER 和 CM_MOUSELEAVE 消息分别在鼠标进入或离开组件的相应区域时触发，类似于 JavaScript 中的 OnMouseOver 和 OnMouseLeave 事件。

实际上，C++ Builder 中存在很多有用的内部消息，例如 CN_MEASUREITEM、CN_DRAWITEM、CM_FONTCHANGED、CM_ENABLEDCHANGED 等等。这些消息完全没有公开，如何获得内部消息的有用信息？通过研究 VCL 源代码可以找到答案。

通过 C++ Builder 的内部消息，问题变得很简单，仅仅需要在消息的响应函数中将继承自 TImage 类的 Picture 属性重新赋值即可。在组件中另一项重要的工作是将 LinkPic 和 Picture 处理成同步改变。

Source TLinkImage 组件的实现代码

```
void __fastcall TLinkImage::SetHoverPic(TPicture *Value)
{
    FHoverPic->Assign(Value);
}
//---------------------------------------------------------------------------
void __fastcall TLinkImage::SetPicture(TPicture *Value)
{   // 此函数同时为 Picture 属性和 LinkPic 属性的 write 方法
    Picture->Assign(Value);
    FLinkPic->Assign(Value);
}
//---------------------------------------------------------------------------
void __fastcall TLinkImage::MouseEnter(Messages::TMessage &Message)
{
    Picture->Assign(FHoverPic);
}
```

```
//---------------------------------------------------------------------------
void __fastcall TLinkImage::MouseLeave(Messages::TMessage &Message)
{
        Picture->Assign(FLinkPic);
}
//---------------------------------------------------------------------------
```

图 13-4 是 TLinkImage 组件的应用示例，它模拟了网页中常用的效果：当鼠标移动到图像上时，图像会被加亮，并且鼠标会变成手状。

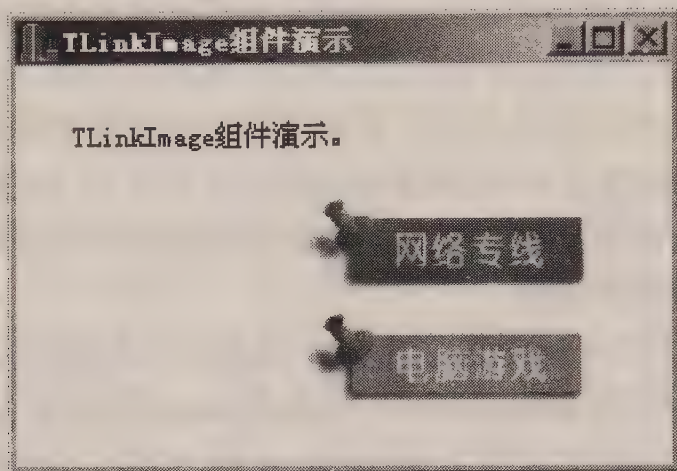


图 13-4　TLinkImage 组件应用示例

这个组件有一定的实际价值。但实际上，将代码稍做改变可以使这个组件工作得更好。例如，完全可以去掉 LinkPic 属性，因为 Picture 已经足够。但这时在组件类的构造函数中将不能够正确地对 FLinkPic 赋值，因为在组件完全建立起来之前，程序不能访问组件的 Picture 属性（我们需要将 Picture 属性赋给 FLinkPic，以替代 LinkPic 属性的作用），这是因为一些主要属性（如 Parent 属性，它包含引用组件的窗体）在组件构建完成前不能正确设置。这时必须重载 CreateWnd 方法，在 CreateWnd 中访问 Picture 属性并对 FLinkPic 赋初值。

我们已经利用了 CM_MOUSEENTER 和 CM_MOUSELEAVE 内部消息，更进一步的工作应该为组件增加 OnMouseEnter 和 OnMouseLeave 两个有用的事件，具体方法在后文中给出。

13.2.3　编写组件代码

1. 访问权限

C++ Builder 扩展了 C++的关键字，其中最为明显的例子就是类定义中的“__published”关键字。这样一个 C++ Builder 类的定义部分包括了四个域 private、protected、public、__published。

这四个域代表了四个级别的访问权限，访问权限让定义什么代码能访问对象的哪一部分。通过描述访问级别，定义了组件的接口。对于某个属性、事件或方法定义在哪个域是很明确的。

表 13-1 描述了四个域的具体含义。

表 13-1 C++ Builder 的作用域及其含义

作 用 域	描 述
Private	私有。隐藏实现细节。除了本类实现中可以访问，其他任何形式访问都被拒绝
Protected	保护。开发者接口。任何实际的应用中无法访问此域中的属性或方法。但此类的继承类可以访问
Public	公有。此域中定义的内容可以在程序运行期间被任意访问
__published	发布。定义设计时接口。此域中定义的属性可显示在 Object Inspector 窗口中，并在设计期间进行修改

2. 创建属性

属性（Property）是组件中特殊的部分，它们支持用户在设计时浏览和操作，并且在交互过程中能立即得到返回结果。C++ Builder 使用新增的__property 关键字来说明一个属性。

属性具有明显的优越性，包括以下四点：

- 用户可以在设计时设置属性。这是非常重要的，因为不像方法只能在运行时访问，属性使用户在运行程序之前就能定制组件（如果这个属性拥有 write 功能的话），通常组件不应包含很多的方法，它们的功能应尽量通过属性来实现。
- 属性能隐藏详细的实现细节。
- 属性能引起简单的赋值之外的响应，如触发事件。
- 属性提供简易的使用方式。组件的属性可以像变量一样简单地赋值取值。

3. 公布继承属性

所有组件都从父类型继承属性。当从已有组件继承时，新组件将继承父类型的所有属性。如果继承的是抽象类，则会继承 protected 或 public 的属性。如想使用户访问 protected 或 public 属性，可以简单的将该属性重定义为__published。例如创建组件从 TWinControl 继承，它继承了 Ctl3D 属性，但是该属性是 protected 的，因此用户在设计和运行时不能访问 Ctl3D，通过在新组件中将 Ctl3D 重声明为__published，改变 Ctl3D 的访问级别，可以使新组件拥有这个属性。

例如下面的一段代码将 Ctl3D 声明为__published。

```
class TNewComponent: public TWinControl
{
    __published:
    __property   Ctl3D;
}
```

我们可以把保护的属性公布出来，但不能做相反的事，即把原有的__published 属性声明为 protected。

4. 添加新属性

添加组件的属性，首先要对属性进行声明。

声明组件的属性，应该描述三个方面的内容：

- 属性名。
- 属性的类型。
- 读取和设置属性值的方法。

至少，组件属性应当定义在组件对象声明的 public 部分，这样才可以在运行时进行访问；为了能在设计时编辑属性，应当将属性在 published 部分声明，这样属性能自动显示在 Object Inspector 窗口中。

一般来说，现实的属性对应一个变量（数据域）存储属性的数据值。通常 C++ Builder 组件遵循下列约定：

- 属性数据存储在对象的数据域处。
- 属性对象数据域的标识符以 F 开头，例如定义在 TControl 中的属性 FWidth。
- 属性数据的对象域应声明在 private 部分。

使属性数据可用的最简单的办法是直接访问。属性声明的 read 和 write 部分描述了怎样不通过调用访问方法来给内部数据域赋值。但一般都用 read 进行直接访问，而用 write 进行方法访问，以改变组件的状态。

虽然属性声明的 read 和 write 部分允许使用数据操作属性值，但更为一般的方式是定义读写方法，读写方法一般以 GetXXX 和 SetXXX 命名。

当声明一个属性，可以选择声明属性的缺省值。设置缺省值采用“default = 缺省值”的方式声明，另一个与设置缺省值相关的关键字是 nodefault。

特别应该注意的是，在窗体设计时也就是 Object Inspector 窗口中出现的值并不是“default= ”声明的缺省值。要定制窗体设计时的“缺省值”应该在组件的构造函数或重载 CreatWnd 函数中对组件赋值。

5. 创建组件的对象方法

方法就是组件类的成员函数。较之属性和事件，方法是组件中真正能执行的一段代码，是组件类真正动作的来源。组件类中的方法的定义与普通的 C++类中方式相同。

C++ Builder 中开发组件的对象方法应该注意：

- 尽量将方法封装到属性内，尽可能利用属性的优点。
- 避免规定某些方法必须按照一定顺序调用。
- 避免调用方法后造成另外一些方法、属性或事件的不可用。

方法中希望组件用户可见的方法在 public 中声明，不希望用户直接调用的方法在 protected 中声明。属性的读写方法通常在 private 中声明，但也可以在 protected 中声明为虚函数，以支持组件用户重载。

13.2.4 创建事件

事件代表了程序运行期的一些行为，如鼠标单击、鼠标移动、定时器到时等。事件是能

够被组件捕获并通知给用户的机制，它将组件发生的事情和用户联系起来。作为组件开发者来说，应该将有价值的系统和用户行为（如鼠标、键盘活动）或组件变化状况（如对话框显示、对话框关闭）作为组件事件提供给使用者。组件使用者根据事件句柄编写自己的代码以便在相应的事件发生时执行相应的动作。从使用者的角度来说，事件是与系统或组件事件有关的名称，用户能给该事件赋特定的函数供调用。

通过事件，组件的使用者可以不关心组件类的细节，利用事件句柄插入任何代码使组件具有不同的行为，但对于组件开发者，必须对事件的捕获，事件的触发，事件的投递相当清楚，因此必须理解下面的概念：

- 事件是指向事件句柄的指针。
- 事件是属性。

事件的本质是指向用户自定义方法的指针，也就是事件句柄指针。通过这个指针，在某些事件发生时就可以找到并执行用户为此事件定义的相应函数。因此，所谓事件其实是一个指针类型的值，而不是一个行为函数。所以事件是可以被赋值的，以使事件指向不同的处理函数，进行不同的处理。例如，在程序设计中经常编写这样的代码：

```
swtich (flag){
case 0:    OnMouseDown = Function1;
           break;
case 1:    OnMouseDown = Function2;
           break;
           ……
}
```

1. 事件是属性

事件是一个值，所以在组件开发中事件被作为一种特殊的属性处理。不像大多数其他属性，事件不使用方法来实现 read 和 write 部分。事件属性使用了相同类型的私有对象域作为属性，按约定域名在属性名前加“F”，例如下面的例子定义了一个 OnClick 事件。

```
class TmyControl : public TCustomControl
{
private:
    TNotifyEvent   FonClick;
    ……
__published:
    __property TNotifyEvent OnClick = { read=FonClick,write=FonClick};
    ……
};
```

2. 创建事件

C++ Builder 中已经定义了许多标准事件封装大部分 Windows 消息，这些事件缺省往往是 protected，这表明组件开发者可以继承并将需要的事件公布给用户使用。标准事件可分为两组：

- 继承于 TControl 类的标准事件。这些事件包括： OnOnCanResize、OnClick、

OnConstrainedResize、OnDblClick、OnDragDrop、OnDragOver、OnEndDock、OnMouseDown、OnMouseMove、OnMouseUp、OnResize、OnStartDock 等，

- 继承于 TWinControl 类的标准事件。这些事件包括：OnEnter、OnExit、OnGetSiteInfo、OnKeyDown、OnKeyPress、OnKeyUp、OnMouseWheel、OnMouseWheelDown、OnMouseWheelUp、OnUndock 等，

以上这些事件定义为 protected 类型，开发者可以继承。在继承组件类中通过在__published 中对事件再次声明，即可视组件拥有相应的事件属性，本章的第一个例子中已经做了这样的工作。

继承标准事件时可以修改自定义组件响应某种标准事件的方法，可以重载事件的内部处理方法并将其赋给事件。事件的内部处理方法命名为事件名除去前面的 On 字，并用“void __fastcall 函数名（void）”的方式声明。例如重载 OnClick 事件的方法：

```
void __fastcall TMyControl::Click(void)
{
        TWinControl::Click();
        // 以下加入自己的代码
        ……
}
```

但仅仅提供标准事件的支持往往是不够的，为组件增加必须的新事件是创建实用组件重要任务。关于定义新事件的具体实现方法，将在本章稍后通过实例介绍。

13.3 创建图像时钟组件

现在开始创建一个较为复杂的组件 TCoolClock，该组件是一个具有多种功能和丰富样式的实时图像时钟。该组件从底层创建，它继承于 TCustomControl 类。使用 TCustomControl 和 TGraphicControl 作为父类对于组件创建来说基本相同，但在这里希望 TCoolClock 组件可以继承 TWinControl 的一些事件和特性，更重要的一点是，在本书第 16 章将会把 TCoolClock 组件转换为一个 ActiveX 控件，能够做这种转换的前提是该组件为 TWinControl 类的某级子类。

自创建的组件 TCoolClock 如图 13-5 所示。

在创建图像时钟组件 TCoolClock 的过程中，我们将接触并学习一些重要概念和组件编程技术：

- 为组件定义枚举类型，并添加枚举类型的属性。
- 重载组件的 Paint 对象方法。该方法提供了组件的用户界面，并在组件重画时自动调用，重载 Paint 对象方法对于从 TCustomControl 和 TGraphicControl 继承的组件至关重要。
- 为组件增加 TPersistent 持久性对象类派生的属性。在 TCoolClock 组件中我们增加了 TPen 类和 TBrush 类的属性，并在组件内部对属性的事件进行了处理。
- 定制全新的事件类型。在 TCoolClock 组件中会演示两种不同类型的新事件创建方法。

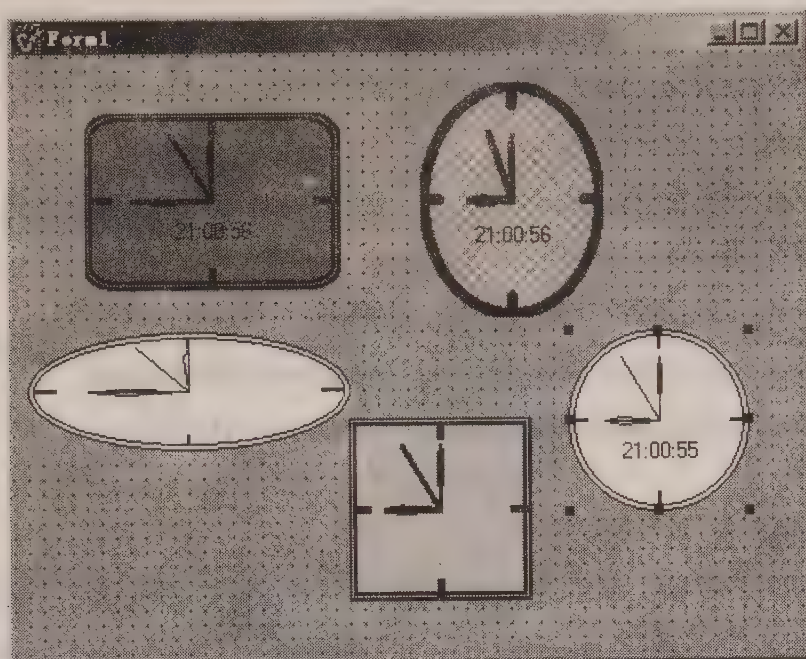


图 13-5　在窗体设计阶段使用自创建组件——TCoolClock

为了对要做的工作有一个整体认识，有必要首先了解一下 TCoolClock 组件最终版本的声明部分，代码如下。

Source　TCoolClock 组件的完整声明

```
enum TClockStyle {csRect,csRoundRect,csEllipse};
class PACKAGE TCoolClock : public TCustomControl
{
private:
    TPen *FPen;
    TBrush *FBrush;
    bool FShowTime;
    TTimer *FTimer;
    TClockStyle FClockStyle;
    TNotifyEvent FClockStep;
    TNotifyEvent FMouseEnter;
    TNotifyEvent FMouseLeave;
    void __fastcall Paint(void);
    void __fastcall SetPen(TPen *Value);
    void __fastcall SetBrush(TBrush *Value);
    bool __fastcall GetActive(void);
    void __fastcall SetActive(bool Value);
    TDateTime __fastcall GetTime(void);
    void __fastcall SetShowTime(bool Value);
    void __fastcall SetClockStyle(TClockStyle Value);
    void __fastcall TCoolClock::DrawPoint(TPoint Point[3],int Pos);
```

```
protected:
        void __fastcall UpdateIt(TObject* Sender);
        void __fastcall ClockStep(void);
        void __fastcall MouseEnter(void);
        void __fastcall MouseLeave(void);
        void __fastcall TimerOnTimer(TObject* Sender);
        void __fastcall CMMouseEnter(Messages::TMessage &Message);
        void __fastcall CMMouseLeave(Messages::TMessage &Message);
public:
BEGIN_MESSAGE_MAP
        MESSAGE_HANDLER(CM_MOUSEENTER,TMessage,CMMouseEnter)
        MESSAGE_HANDLER(CM_MOUSELEAVE,TMessage,CMMouseLeave)
END_MESSAGE_MAP(TCustomControl)
        __fastcall TCoolClock(TComponent* Owner);
        __fastcall ~TCoolClock(void);
__published:
        __property Width={default=100};
        __property Height={default=100};
        __property Align;
        __property ParentColor;
        __property PopupMenu;
        __property ShowHint;
        __property Visible;
        __property OnClick;
        __property OnDblClick;
        __property OnKeyDown;
        __property OnKeyPress;
        __property OnKeyUp;
        __property OnMouseDown;
        __property OnMouseMove;
        __property OnMouseUp;
        __property bool Active={read=GetActive, write=SetActive, default=true};
        __property TPen *Pen={read=FPen,write=SetPen};
        __property TBrush *Brush={read=FBrush,write=SetBrush};
        __property TDateTime CurrentTime={read=GetTime};
        __property bool ShowTime={read=FShowTime,write=SetShowTime};
        __property TClockStyle ClockStyle=
            {read=FClockStyle,write=SetClockStyle,default=csEllipse};
        __property TNotifyEvent OnClockStep={read=FClockStep,write=FClockStep};
        __property TNotifyEvent OnMouseEnter={read=FMouseEnter,write=FMouseEnter};
```

```
        __property TNotifyEvent OnMouseLeave={read=FMouseLeave,write=FMouseLeave};
};
```

13.3.1 为组件增加枚举类型属性

TCoolClock 组件实现三种不同的时钟样式，通过枚举类型属性控制时钟样式较为合适。在 TCoolClock 组件中，我们创建了一个枚举类型的属性 ClockStyle，该属性描述时钟组件的三种样式——圆形（椭圆形）、矩形、圆角矩形。为此需要首先定义一个枚举类型 TClockStyle：

```
enum TClockStyle {csRect,csRoundRect,csEllipse};
```

使用该枚举类型定义 ClockStyle 的数据域私有变量 FClockStyle，ClockStyle 属性直接从相应的数据域读取，并创建属性的写方法 SetClockStyle，该方法代码如下：

```
void __fastcall TCoolClock::SetClockStyle(TClockStyle Value)
{
        if(FClockStyle!=Value){
                FClockStyle=Value;
                Invalidate();
        }
}
```

注意 ClockStyle 属性是会改动组件对象外观，所以在对数据域赋值后一定要调用 Invalidate 函数重画时钟组件。因为要进行重画，所以应该判断属性是否真的改变，如为真，才要求系统重画组件，否则将跳过代码，方法马上结束。这样做有助于提高组件效率，对于设置属性后要进行复杂操作的写方法一般都应这样。

此外，组件类中定义了 ShowTime 布尔类型的属性和 CurrentTime 属性，ShowTime 属性决定是否显示文字时间。它的实现较为简单，此属性同样直接从数据域读取，其写方法实现如下：

```
void __fastcall TCoolClock::SetShowTime(bool Value)
{
        if (FShowTime!=Value){
                FShowTime=Value;
                Invalidate();
        }
}
```

CurrentTime 属性为一个只读属性，所以在 Object Inspector 中无法看到这个属性。CurrentTime 属性简单的返回系统当前时间。

```
TDateTime __fastcall TCoolClock::GetTime(void)
{
        return Time();
}
```

另外一个简单的属性是 Active，此属性并没有实际的数据域与之对应，它通过读写方法与内建时钟对象的 Enabled 属性对应。此方法的实现与本章第一个范例中的 Active 属性实现相同。

图 13-6 列出了 TCoolClock 组件的一些属性。

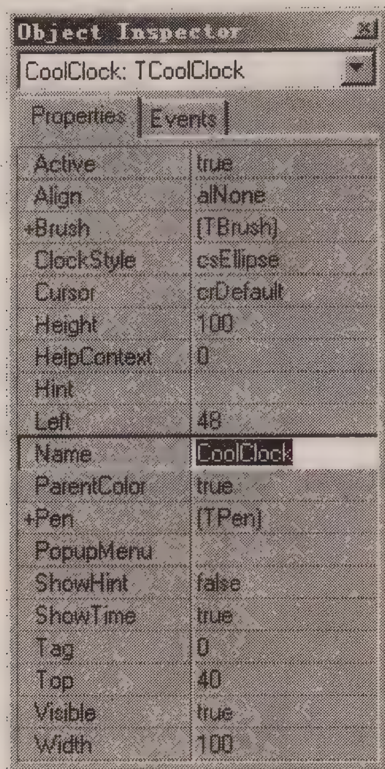


图 13-6　TCoolClock 组件的属性

在创建完成属性后，应该注意在组件类的构造函数中设置属性的缺省值。

```
__fastcall TCoolClock::TCoolClock(
    TComponent* Owner): TCustomControl(Owner)
{
    Width=100;
    Height=100;
    FShowTime=true;
    FTimer=new TTimer(this);
    FTimer->Interval=1000;
    FTimer->Enabled=true;
    FTimer->OnTimer=TimerOnTimer;
    ClockStyle=csEllipse;
}
```

图像时钟组件是一个动态的组件，它每一秒重画时钟一次，而这一切是由组件的内建定时器对象 FTimer 驱动的。

13.3.2　绘制时钟

从 TCustomControl 和 TGraphicControl 继承的组件存在 Paint 对象方法。此方法在组件重

画时被自动调用。这里所进行的一切绘制组件的操作都应该在重载 Paint 对象方法中执行。

TCustomControl 和 TGraphicControl 类提供了一个保护的 Canvas 属性，借助这个属性可以方便地完成绘制组件的工作。

以下是图像时钟组件的关键代码部分，重载的 Paint 对象方法。

Source TCoolClock 组件的时钟图像绘制

```
//---------------------------------------------------------------------------
void __fastcall TCoolClock::Paint(void)
{
        // 绘制时钟边框
        switch (FClockStyle){
        case csEllipse:
                Canvas->Ellipse(0,0,Width,Height);
                Canvas->Ellipse(3,3,Width-3,Height-3);
                break;
        case csRect:
                Canvas->Rectangle(0,0,Width,Height);
                Canvas->Rectangle(3,3,Width-3,Height-3);
                break;
        case csRoundRect:
                Canvas->RoundRect(0,0,Width,Height,Width/4,Height/4);
                Canvas->RoundRect(3,3,Width-3,Height-3,Width/4-3,Height/4-3);
 }
        // 绘制表盘刻度
        Canvas->Pen->Width=Canvas->Pen->Width*2;
        Canvas->MoveTo(Width/2,4);
        Canvas->LineTo(Width/2,4+Height/15);
        Canvas->MoveTo(Width/2,Height-4);
        Canvas->LineTo(Width/2,Height-Height/15-4);
        Canvas->MoveTo(4,Height/2);
        Canvas->LineTo(4+Width/15,Height/2);
        Canvas->MoveTo(Width-4,Height/2);
        Canvas->LineTo(Width-Width/15-4,Height/2);
        Canvas->Pen->Width=Canvas->Pen->Width/2;
        // 绘制时针、分针、秒针
        float Len=((Width<Height)?Width:Height);
        unsigned short h,m,s,ms;
        Time().DecodeTime(&h,&m,&s,&ms);
        TPoint Point[3];
```

```
        // 绘制分针;
        Point[1].x=Width/2;Point[1].y=Height/2-Len*7/20;
        Point[0].x=Width/2-Len/100;Point[0].y=Height/2-Len/3.5;
        Point[2].x=Width/2+Len/100;Point[2].y=Height/2-Len/3.5;
        DrawPoint(Point,m);
        // 绘制秒针;
        Point[1].x=Width/2;Point[1].y=Height/2-Len*8/20;
        Point[0].x=Width/2;Point[0].y=Height/2;
        Point[2].x=Width/2;Point[2].y=Height/2;
        DrawPoint(Point,s);
        // 绘制时针;
        Point[1].x=Width/2;Point[1].y=Height/2-Len*6/20;
        Point[0].x=Width/2-Len/50;Point[0].y=Height/2-Len/5;
        Point[2].x=Width/2+Len/50;Point[2].y=Height/2-Len/5;
        DrawPoint(Point,h*5);
        // 输出精确时间（文字）;
        if (FShowTime){
        TRect TextRect=Rect(Width/4,Height*3/5,Width*3/4,Height*4/5);
                DrawText(Canvas>Handle,TimeToStr(Time()).c_str(),
                TimeToStr(Time()).Length(),&TextRect,DT_TOP|DT_CENTER);
        }
    }
    //---------------------------------------------------------------------------
```

绘制图像时钟是本组件中最为复杂的一部分，它需要经过一系列数学运算和绘制，同时在计算方面用到了图形变换的知识。

具体的绘制工作代码与本章的主题没有太大联系，但为了使读者更好地理解程序的运作，在此作简单的解释。

首先做的工作是绘制钟表的边框，绘制哪种类型的边框由组件的 ClockStyle 属性决定。

之后做的工作是绘制表盘刻度，这里只绘制了表盘上下左右四个刻度，注意绘制刻度的大小和位置必须根据表盘的宽和高计算得到，之后绘制的表针也是如此。

再后进行三个表针的绘制工作。本例中将每个表针处理为一个菱形，原点是表盘中心，这是容易确定的，而其他三个点由当前时间和钟表的宽度、高度共同决定。在本例中首先根据钟表的尺寸计算出表针在 12 点方向上三个关键点的位置，然后交由 DrawPoint 函数进行旋转计算并在正确位置将表针绘制出来。

DrawPoint 函数实现如下。

Source 时钟刻度绘制

```
    //---------------------------------------------------------------------------
    void _fastcall TCoolClock::DrawPoint(TPoint Point[3],int Pos)
```

```
{
    TPoint Ori,Tmp;
    const double Pi=3.1415926535;
    float Ratio=float(Width)/Height;
    Ori.x=Width/2;Ori.y=Height/2;
    Canvas->MoveTo(Ori.x,Ori.y);
    for (int i=0;i<3;i++)
    {
        Tmp.x=((Point[i].x-Ori.x)*cos(Pos*Pi/30)+(-Point[i].y+Ori.y)*sin(Pos*Pi/30))*Ratio+Ori.x;
        Tmp.y=(Point[i].x-Ori.x)*sin(Pos*Pi/30)+(Point[i].y-Ori.y)*cos(Pos*Pi/30)+Ori.y;
        Canvas->LineTo(Tmp.x,Tmp.y);
        if (i==1) {
            Canvas->MoveTo(Ori.x,Ori.y);
            Canvas->LineTo(Tmp.x,Tmp.y);
        }
    }
    Canvas->LineTo(Ori.x,Ori.y);
}
//---------------------------------------------------------------------------
```

最后进行的是绘制文字时间的工作，因为需要将文字绘制在表盘的中心，所以 Tcanvas 的 TextOut 方法已经不能满足需要，在这里调用了 API 函数 DrawText，为输出文字定制一个矩形框，并让文字位于这个矩形框的中心。DrawText 函数的声明如下：

```
int DrawText(
    HDC hDC, // 图形设备描述表句柄
    LPCTSTR lpString,    // 指向绘制字符串的指针
    int nCount, // 绘制字符的个数
    LPRECT lpRect, // 确定绘制文本的矩形框
    UINT uFormat    // 绘制标志变量
  );
```

图像时钟绘制出的样子如图 13-7 所示。

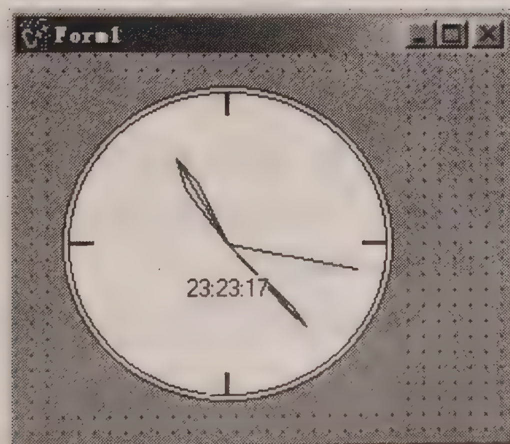


图 13-7　绘制完成的时钟

13.3.3 增加 TPersistent 属性

为了提供对图像时钟样式的更多控制，需要为 TCoolClock 组件添加画笔和画刷两个继承自 TPersistent 类的属性，对于这种持久性对象，必须手工建立和解除内部对象，这一点与内建的 TTimer 对象相同（实际上 TTimer 对象也是继承于 TPersistent 类的对象）。

因为画笔和画刷的改变会造成图像的改变，所以要在程序中处理画笔和画刷的 OnChange 事件。

画笔和画刷同样需要建立内部数据域，提供属性的读取。

在定义部分增加：

```
private:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall SetPen(TPen *Value);
    void __fastcall SetBrush(TBrush *Value);
protected:
    void __fastcall UpdateIt(TObject* Sender);
    __published:
    __property TPen *Pen={read=FPen,write=SetPen};
    __property TBrush *Brush={read=FBrush,write=SetBrush};
```

在构造函数中创建对象，并处理它们的 OnChange 事件。

```
    FPen=new TPen();
    FBrush=new TBrush();
    FPen->OnChange=UpdateIt;
    FBrush->OnChange=UpdateIt;
```

UpdateIt 方法简单的告诉系统重画组件。

```
void _fastcall TCoolClock::UpdateIt(TObject* Sender)
{
    Invalidate();
}
```

因为 FPen 和 FBrush 是指针变量，在写方法中不能使用等号进行简单的赋值，而应该使用 TPersistent 类的 Assign 方法进行赋值，这与本章第二个实例处理 TPicture 类属性的方式相同。

画笔和画刷的写方法实现如下：

```
void __fastcall TCoolClock::SetPen(TPen *Value)
{
    FPen->Assign(Value);
    Invalidate();
}
```

```
void __fastcall TCoolClock::SetBrush(TBrush *Value)
{
        FBrush->Assign(Value);
        Invalidate();
}
```

因为画笔、画刷属性应用于实际绘图，所以在 Paint 方法中应处理组件的 Canvas 的 Pen 和 Brush 属性，使之与组件的画笔和画刷具有相同的值。

```
void __fastcall TCoolClock::Paint(void)
{
        // 使用组件的画笔和画刷属性
        Canvas->Pen=FPen;
        Canvas->Brush=FBrush;
        ......
```

图 13-8 显示了使用画笔画刷后绘制的时钟。

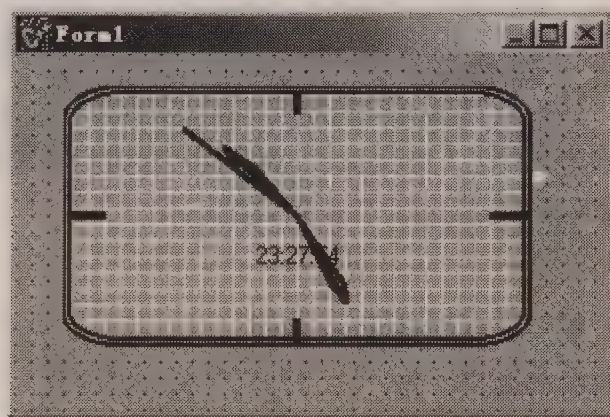


图 13-8　使用了特殊画笔和画刷绘制的图像时钟

13.3.4　增加新创建的事件

创建新事件比继承标准事件复杂，一般来说，创建一个全新事件应遵循以下步骤：

(1)　声明一个事件类型字段。
(2)　声明事件。
(3)　定义事件处理函数。
(4)　触发事件。

在图像时钟组件中我们新建了 OnClockStep 事件，这个简单的事件在每次时钟走一步时发生。首先我们为这个事件声明一个事件类型字段，这相当于一般属性的数据域。

```
TNotifyEvent FClockStep;
```

这里使用了 TnotifyEvent 类型，这种事件类型的特点是只有一个参数 Sender。这种事件类型应用于很多标准事件，例如 OnTimer 和 OnClick。TNotifyEvent 类型定义如下：

```
typedef void __fastcall (__closure *TNotifyEvent)(System::TObject* Sender);
```

当然也可以使用像 TkeyEvent 这样具有多个参数的事件类型，甚至可以自己定义事件类

型，但注意定义的事件类型必须以__closure 关键字说明。

下面的工作是声明事件，声明部分在__published 中进行。

```
__property TNotifyEvent OnClockStep={read=FClockStep,write=FClockStep};
```

按照命名规定，所有事件都应以 On 字开头，而事件的处理函数应以去除 On 字的事件名命名。定义事件的处理函数用于激活事件类型字段 FclockStep 指向的函数。这里必须考虑的问题是事件句柄可能为 NULL，而这时不能再激活事件类型字段指向的函数。

事件处理函数如下：

```
void __fastcall TCoolClock::ClockStep(void)
{
    if (FClockStep)
    {
        FClockStep(this);
        // 此处可以加入特定的操作
    }
}
```

最后要处理的是触发事件，也就是决定新定义的事件何时发生。对于创建新事件来说这是关键的一步。对于范例中的 OnClockStep 事件来说，它是与相应定时器紧密相关的。在定时器事件响应函数 TimerOnTimer 中执行 OnClockStep 事件的处理函数。

```
void __fastcall TCoolClock::TimerOnTimer(TObject* Sender)
{
    ClockStep();            // 触发 OnClockStep 事件
    UpdateIt(NULL);         // 时钟前进一步
}
```

在图像时钟组件中还创建了另外两个有用的事件 OnMouseEnter 和 OnMouseLeave。这两个事件分别在鼠标移动到组件上和鼠标从组件移出时触发。建立这两个事件利用了 C++ Builder 的内部消息。但其基本实现步骤和 OnClockStep 事件是相同的。

完成 OnMouseEnter 和 OnMouseLeave 事件唯一的难点在于确定事件何时激活。在组件类的声明部分建立了消息映射：

```
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(CM_MOUSEENTER,TMessage,CMMouseEnter)
    MESSAGE_HANDLER(CM_MOUSELEAVE,TMessage,CMMouseLeave)
END_MESSAGE_MAP(TCustomControl)
```

在这两个消息的处理程序中调用事件的处理过程，以触发事件。

```
void __fastcall TCoolClock::CMMouseEnter(Messages::TMessage &Message)
{
    MouseEnter();
}
void __fastcall TCoolClock::CMMouseLeave(Messages::TMessage &Message)
{
    MouseLeave();
```

}

组件安装完成后，创建的事件会自动显示在 Object Inspector 的事件页中，如图 13-9 所示，可以看到其中已经有了新创建的事件 OnClockStep、OnMouseEnter 和 OnMouseLeave。

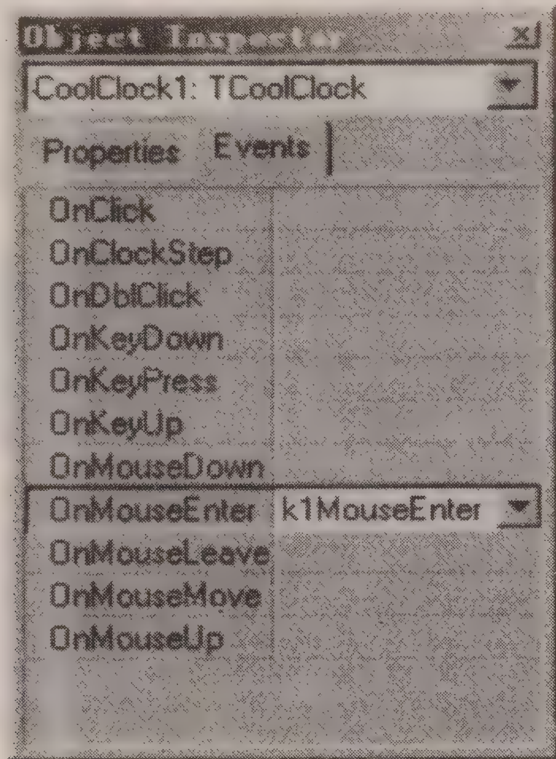


图 13-9　TcoolClock 组件的事件页

13.3.5　组件面板位图

在安装 TcoolClock 组件之前，必须为 TcoolClock 组件创建一个合适的位图，此位图用于在组件选项板上显示。如果不这样做，C++ Builder 会自动使用父类的位图，如果父类并非一个可用的组件（例如 TCustomEdit），则会使用一个缺省位图，如图 13-10 所示。

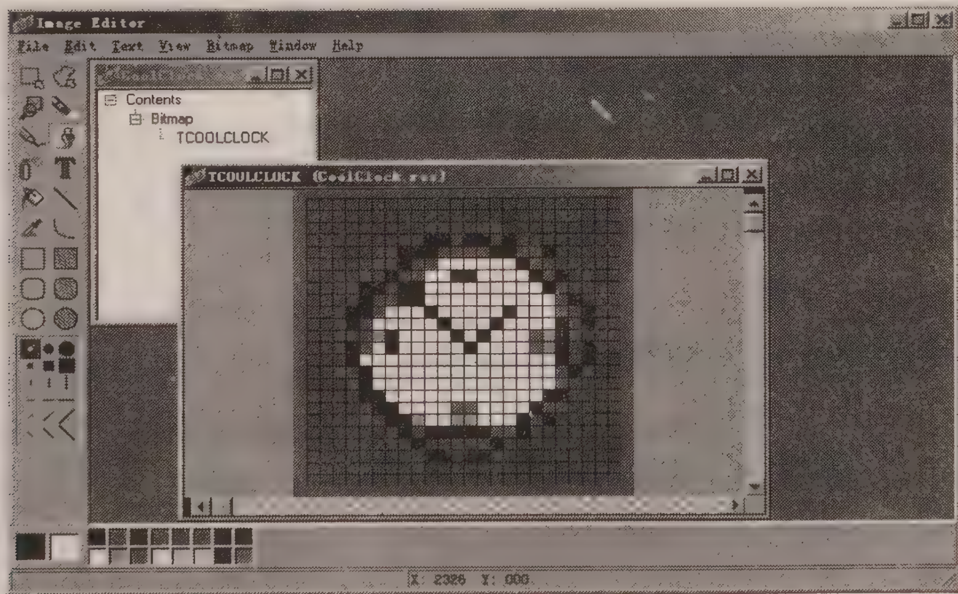


图 13-10　在 Image Editor 中创建组件面板位图

为组件创建位图必须遵循一定的规则，这些规则包括：

- 组件位图应包含在一个与组件单元名同名（当然不包括扩展名）的.RES 资源文件

中，例如在本例中资源文件名为 CoolClock.res。

- 位图资源名称必须与组件类名同名，并且字母必须大写。在本例中，位图资源的名称为 TCOOLCLOCK。
- 位图资源必须为 24×24 像素大小。
- 向组件包中加入资源文件，资源文件必须先于组件单元文件加入，否则会出现“The project already contains a form or a moudle named XXXX”的错误信息。

图 13-11 显示了组件包项目编辑器的情况，组件包中包含了组件单元文件和资源文件。

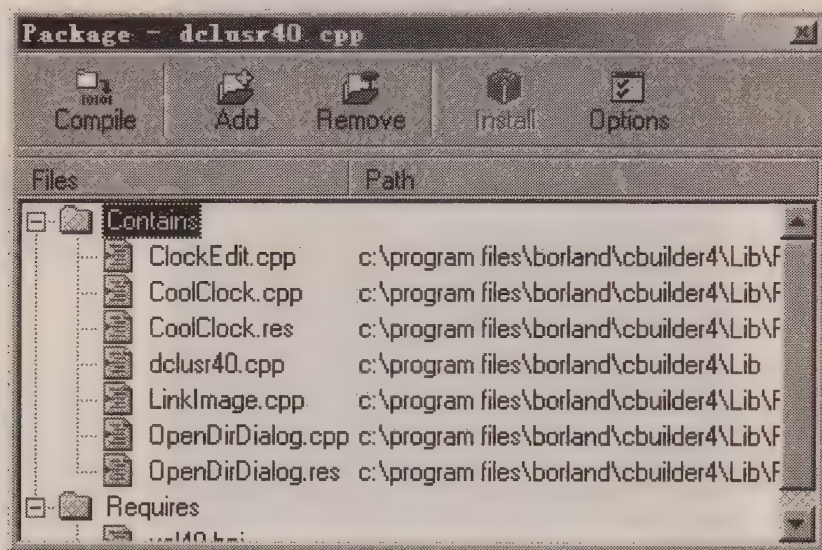


图 13-11　向组件包项目加入单元文件和资源文件

13.3.6　测试 TCoolClock 组件

创建 VCL 组件的一个重要的好处就是可以在组件没有安装时进行调试。其方法与在运行期间实例化一个组件对象的方法相同。在这里，我们编写一个简单范例，用于测试安装后的 TCoolClock 组件。

如图 13-12 所示，ClockTest 范例允许改动图像时钟组件的大部分属性，并且测试时钟组件的两个新增事件 OnMouseEnter 和 OnMouseLeave。

更改时钟颜色的代码为：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    ColorDialog1->Color=CoolClock->Brush->Color;
    if (ColorDialog1->Execute()){
        CoolClock->Brush->Color=ColorDialog1->Color;
        Shape1->Brush->Color=ColorDialog1->Color;
    }
}
```

响应新增的 OnMouseEnter 事件的函数为：

```
void __fastcall TForm1::CoolClockMouseEnter(TObject *Sender)
{
```

```
    StatusBar1->Panels->Items[0]->Text="OnMouseEnter 事件发生!";
}
```

可以看到，虽然 TCoolClock 组件是从底层创建的 VCL 组件，但该组件的使用方式与其他已有 VCL 组件并没有任何分别。

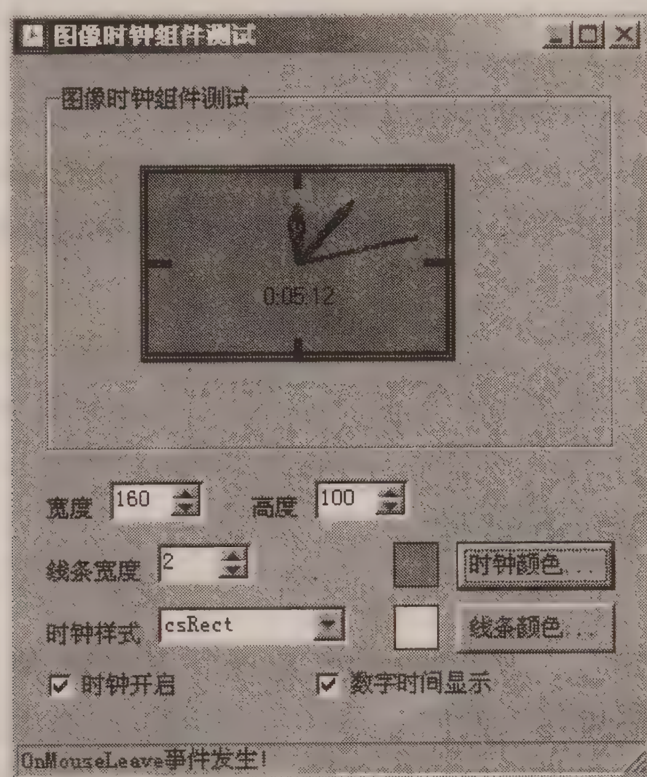


图 13-12　运行时的 ClockTest 程序

13.4　创建非可视化组件

本节将讨论 C++ Builder 中组件编程的另一重要方面——创建非可视化组件。很多强大的功能往往通过封装成非可视化组件完成。例如 C++ Builder 4 提供的非可视化组件实例 TrayIcon，利用该组件可以方便地创建具备指示区图标的应用程序。

可以将任何可以重用的处理过程或具有某种功能的函数封装成非可视化组件（例如 TrayIcon 组件），另外，将具有特定用处的复杂窗体封装成组件以实现窗体重用也是创建非可视化组件的典型例子。

读者可能会对“把窗体封装成非可视化组件”有所疑问：窗体是一个典型的可视化对象，任何一个窗体都直接继承于 TWinControl 类而非 TComponent 类。实际上封装成非可视化组件的对象并非窗体本身，而是创建并显示窗体的过程（或代码）。真正的窗体作为组件类的内部对象被定义。例如，这类组件一般提供 Execute 方法（与 C++ Builder 的标准对话框组件的风格保持一致），其典型的结构为：

```
bool __fastcall TNewFormDialog::Execute(void)
{
    TNewForm *NewForm=new TNewForm(this);
    // 根据 TNewFormDialog 组件的属性值初始化窗体对象中组件
    ............
```

```
        if(NewForm->ShowModal==ID_OK)
        {
            // 根据窗体对象中组件的值更新 TNewFormDialog 组件的属性
            ............
            delete NewForm;
            return true;
        }
        else {
            delete NewForm;
            return false;
        }
    }
```

13.4.1 创建 TOpenDirDialog 组件

本节创建的非可视化组件 TOpenDirDialog 是一个 Win32 风格的选择目录对话框组件，这个对话框组件有较高的实用价值，同时它与 C++ Builder 提供的几个标准对话框组件有更多相似之处：选择目录对话框并非是自己定义的一个窗体，而是利用 Win32 的外壳 API 接口显示标准选择目录对话框。

如前所述，非可视化组件继承于 TComponent 类。在这个组件中，我们模仿 C++ Builder 标准对话框组件，创建了 DirectoryName、Title、InitialDir 三个属性以及 OnClose 和 OnShow 两个事件，并提供 Execute 方法执行目录选择对话框。TOpenDirDialog 声明部分如下：

Source　非可视组件 TOpenDirDialog 的声明

```
//---------------------------------------------------------------------------
class PACKAGE TOpenDirDialog : public TComponent
{
private:
        AnsiString FDirectoryName;
        AnsiString FTitle;
        AnsiString FInitDir;
        TNotifyEvent FOnClose;
        TNotifyEvent FOnShow;
        void __fastcall SetTitle(AnsiString Value);
        void __fastcall SetInitDir(AnsiString Value);
protected:
        void __fastcall DialogClose(void);
        void __fastcall DialogShow(void);
```

```
public:
        __fastcall TOpenDirDialog(TComponent* Owner);
        bool __fastcall Execute(void);
__published:
        __property AnsiString DirectoryName={read=FDirectoryName,write=FDirectoryName};
        __property AnsiString Title={read=FTitle,write=SetTitle};
        __property AnsiString InitialDir={read=FInitDir,write=SetInitDir};
        __property TNotifyEvent OnClose={read=FOnClose,write=FOnClose};
        __property TNotifyEvent OnShow={read=FOnShow, write=FOnShow};
};
//---------------------------------------------------------------------------
```

该范例的大多数代码在 Execute 对象方法中，实现标准目录选择对话框时使用了 SHBrowseForFolder API 函数，关于这个函数的使用已在本书第 3 章 ShellAPI 部分讲述过。

Source TOpenDirDialog 组件的实现代码

```
//---------------------------------------------------------------------------
bool __fastcall TOpenDirDialog::Execute(void)
{
        BROWSEINFO brInfo;
        char *Buffer;
        WideString Root=WideString(InitialDir);
        LPITEMIDLIST RootItemIDList, ItemIDList;
        LPMALLOC ShellMalloc;
        LPSHELLFOLDER DesktopFolder;
        DWORD Eaten, Flags;
        ZeroMemory(&BrInfo,sizeof(brInfo));
        if (SHGetMalloc(&ShellMalloc) == S_OK&&ShellMalloc!=NULL){
                Buffer =(char*) ShellMalloc->Alloc(MAX_PATH);
                SHGetDesktopFolder(&DesktopFolder);
                DesktopFolder->ParseDisplayName(Application->Handle,NULL,
                        Variant(Root), &Eaten, &RootItemIDList, &Flags);
                brInfo.hwndOwner = Application->Handle;
                brInfo.pidlRoot = RootItemIDList;
                brInfo.pszDisplayName = Buffer;
                brInfo.lpszTitle = FTitle.c_str();
                brInfo.ulFlags = BIF_RETURNONLYFSDIRS;
                DialogShow();// 实现目录选择对话框的 OnShow 事件
                ItemIDList = SHBrowseForFolder(&brInfo);
                DialogClose();// 实现目录选择对话框的 OnClose 事件
```

```
            if (ItemIDList!=NULL){
                SHGetPathFromIDList(ItemIDList, Buffer);
                ShellMalloc->Free(ItemIDList);
                FDirectoryName = Buffer;
                return true;
            }
            else {
                ShellMalloc->Free(Buffer);
                return false;
            }
        }
    }
    //---------------------------------------------------------------------------
```

这里比较特别的地方是 DialogShow 和 DialogClose 两行，这两个函数分别是 OnShow 和 OnClose 事件的处理函数，它们通过加载在组件执行代码的关键位置实现组件事件。

13.4.2 使用非可视化组件

在创建完成非可视化组件后，应该为组件提供位图标识。虽然这项工作并非必须的，但确实很有必要。因为对于非可视化组件来说，设计时只会以组件的标识位图显示在窗体上（图 13-13 所示）。具有意义的位图会对设计工作有良好的影响，而使用缺省位图会造成不必要的混乱。

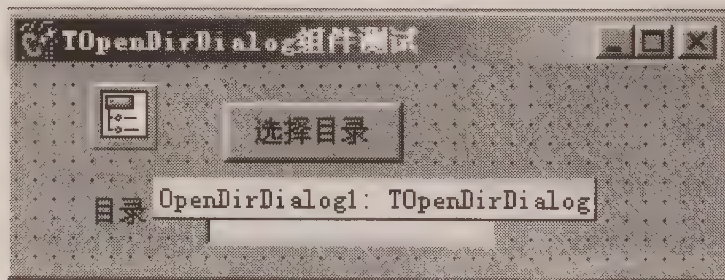


图 13-13 TOpenDirDialog 组件测试程序窗体

现在编写几行代码来测试这个组件。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    OpenDirDialog1->Title="选择目录对话框组件测试";
    if (OpenDirDialog1->Execute())
    Edit1->Text=OpenDirDialog1->DirectoryName;
}
```

TOpenDirDialog 组件具有实用价值和良好的风格。该组件设计成与 C++ Builder 标准对话框组件相同的使用方式，其效果如图 13-14 所示。

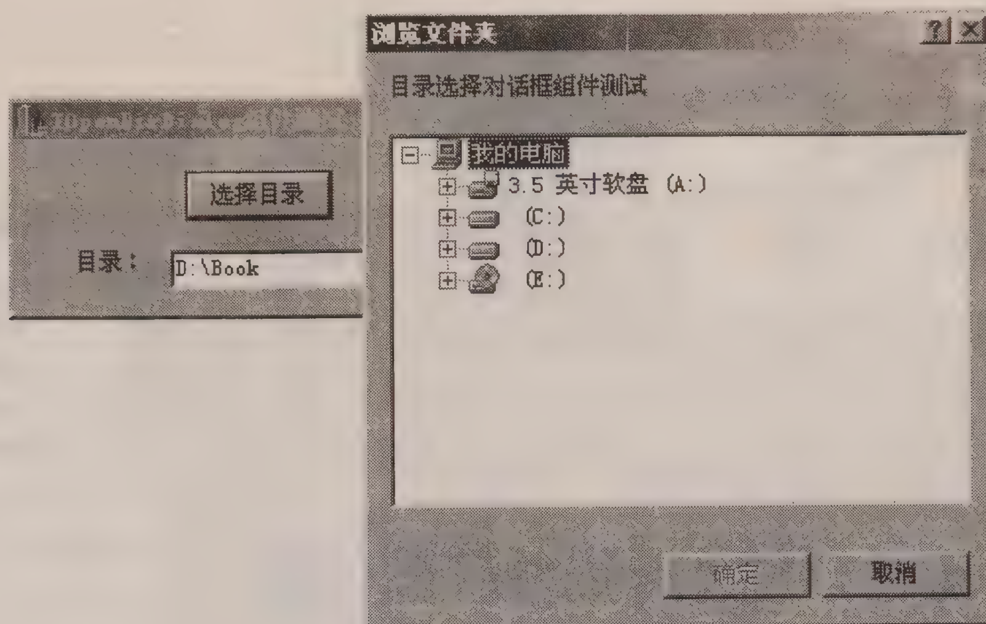
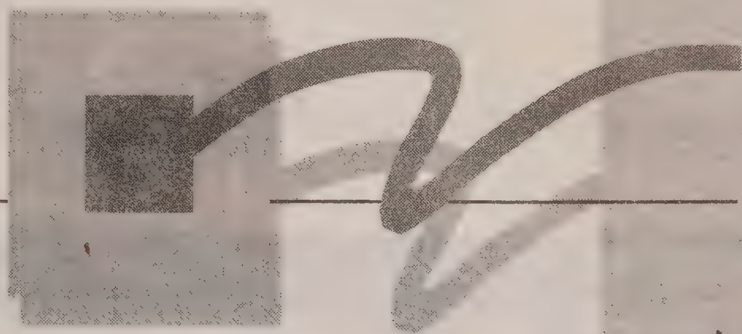


图 13-14　执行过程中的选择目录对话框

第14章

系统环境监视程序
——DLL及应用Windows API编程

动态链接库（DLL）技术
实现指示区图标支持
特殊风格的自绘标题栏
窗口与程序
系统与设备



本章导读

Windows 系统绝大部分是用 C 语言写成的，Windows API 支持 C/C++语言的直接调用。能够毫无障碍地使用 Windows API 是 C++ Builder 较之 Visual Basic、Delphi、PowerBuilder 等可视化应用程序开发工具的重要优势。Windows API 非常强大，它无所不包。一些非常复杂的功能通过 Windows API 可以轻易实现。

本章将完成一个功能强大的“系统环境监视程序”，该程序是具备指示区图标的应用程序，这是依靠 Windows API 实现的。此外，程序中的每一个功能模块，包括特殊风格的界面、查看当前顶级窗口、动态汉化窗口菜单、查看驱动器信息、查看内存资源信息、查看/调整显示状态、查看/设置系统环境变量、查看正在运行的程序信息、杀除进程等等无不是借助了 Windows API 的强劲火力。通过实现这个程序，读者将了解调用 Windows API 的各种方式、以及几个重要的 Windows API 技巧和许多有用的 API 函数。

本章内容包括：

- 关于动态链接库（DLL）技术的讨论，讲述在 C++ Builder 中如何创建和使用 DLL。
- 创建实例程序——系统环境监视程序，实现任务栏图标功能和特殊的界面风格（位于窗口左侧的标题栏）。
- 实现系统环境监视程序的窗口与程序功能模块，包括：查看当前顶级窗口、动态汉化菜单、查看正在运行的程序信息、杀除进程、查看/杀除系统启动程序等等。
- 实现系统与设备功能模块，包括：查看驱动器信息、查看内存资源信息、查看设备信息/调整显示状态、查看/设置系统环境变量等等。

14.1 关 于 DLL

Windows API（Application Programming Interface，应用编程接口）从程序开发人员的角度来看，可以认为是一组用于 Windows 编程的函数库，这些函数实际上存在于一系列重要的 DLL（动态链接库）中，例如 Kernel32.dll、Gdi32.dll、User32.dll、Shell32.dll 等，程序员可以在自己的程序中调用这些函数，同时这些函数也被 Windows 系统本身调用。

除了基本的 Windows API 之外，其他各种 API 调用也都是基于 DLL 的，如 OpenGL、DirectX 等等。而且，C++ Builder 的组件和代码一般也是依赖 DLL 和不同扩展名的动态链接库 BPL（当然也可以依赖静态链接）。上一章讨论了 C++ Builder 的组件包，基本建立的也是 DLL。另外，在第 13 章中创建的高效率的 ISAPI 程序也是基于 DLL 的。

DLL，即 Dynamic Linked Libraries（动态链接库），是随 Windows 系统出现的新的机制，一般来说，DLL 中包含一些其他程序可以调用的过程和函数的集合，也可以包含可被其他程序利用的资源。DLL 动态链接较之以往的静态链接更有效地利用了资源，它具有以下重要优点：

- 如果不同的程序使用相同的 DLL，只需将 DLL 在内存中装载一次，这种做法大大节省了系统资源。DLL 映射到每个进程的专用地址空间中，如果手头有一些实现复杂功能的例程，或者一些复杂的窗体，它们是多个应用程序所需要的，这时，将它们放在一个 DLL 中是非常不错的选择。这样，在同时运行多个使用该 DLL 的程序时，可以减小程序的规模，进而节省内存。
- 可以提供 DLL 的不同版本，代替原有的 DLL。在新的 DLL 中可以包含新的函数过程，并且可以更新原有函数，只要该函数的参数和原版本 DLL 中的函数具有相同的参数，则无需重新编译使用该 DLL 的应用程序，就可以正常运行。这个优点适用于复杂的而且需要不断更新或改正错误的应用程序，可以将程序化分为多个 DLL，这样只需对需要改变的部分进行操作，而不用对整个大的执行文件进行改动。
- DLL 中可以嵌入资源。例如，可以建立 DLL 的不同版本来保存不同语言的字符串，对于应用程序的不同语言版本，只需替换相应的 DLL 即可。这对开发不同语言版本的应用程序特别重要。C++ Builder 提供了 Resource DLL 向导支持方便的创建资源 DLL。
- DLL 独立于编程语言，因为它们已经进行了编译。这就意味着，我们在 C++ Builder 中创建了一个 DLL，而在其他的 Delphi、Visual Basic、Visual C++等等开发工具中都可以调用它。

在 C++ Builder 环境中，使用和创建 DLL 都很方便。下面将简单介绍在 C++ Builder 中创建和调用 DLL 的方法。

14.1.1 在 C++ Builder 中创建 DLL

C++ Builder 为创建 DLL 提供了一个框架。和创建一般的应用程序相同，选择【**File/New**】，打开 NewItem 对话框，然后选择 DLL 项目（如图 14-1 所示）。

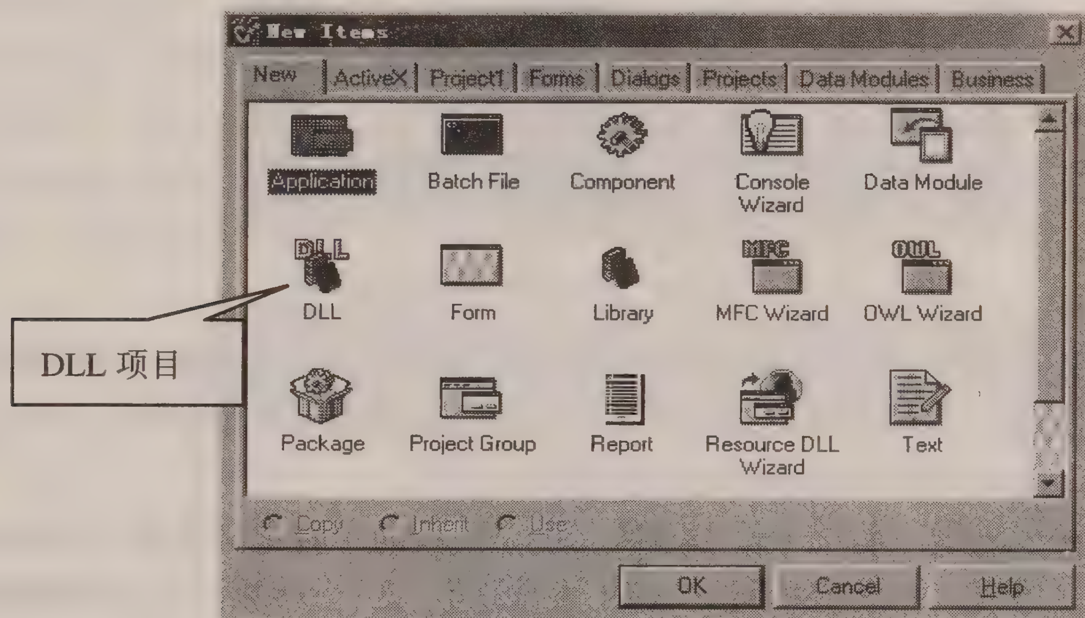


图 14-1 在 C++ Builder 中创建一个 DLL

之后 C++ Builder 会产生一个 DLL 工程，与其他应用程序工程不同，它只产生一个 Project.cpp 文件，该文件包含一个 DLL 的进入点函数 DLLEntryPoint，程序大致如下：

```
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    return 1;
}
```

下一章将对 DLL 进行具体深入的分析。现在首先来创建一个具有实际意义的 DLL 范例程序，这样可使读者对 DLL 有感性的认识。

C++ Builder 在可视化创建窗体方面具有优势，使用 DLL 可以容易地实现窗体重用，也就是说，可以在 C++ Builder 中创建完整窗体并放在 DLL 中。这样在其他的开发环境中（比如 Visual C++）就可以使用这个窗体。本节将创建通过 DLL 实现窗体重用的范例，该 DLL 提供实现选择目录功能的 VCL 窗体。

在建立了 DLL 框架后，选择【**File/New Form**】创建一个窗体，以向窗体中加入需要的组件。本范例将创建一个选择目录的可重用窗体，创建好的窗体如图 14-2 所示。

窗体中的 Label1 组件是留给调用该窗体的程序提供提示信息的，我们设定几个文件操作组件的连结关系，并且填写 TDirectoryListBox 组件的 OnChange 事件，使选择的目录在文本编辑框中显示出来。

然后可以在 DLL 中加入输出函数（Export Function），该函数可以在其他应用程序中调用。首先定义一个输出函数：

```
extern "C" __declspec(dllexport)
bool _stdcall SelectDirectory(LPCTSTR Title,LPCTSTR Msg,LPTSTR Dir);
```

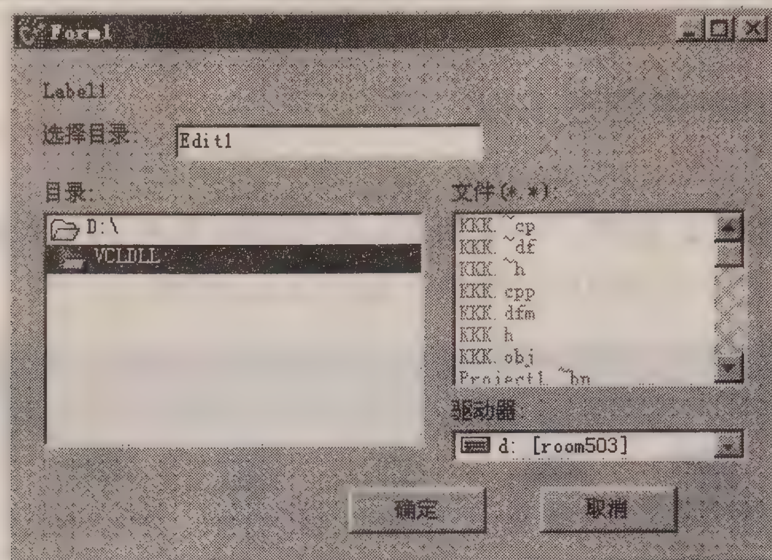


图 14-2　在 C++ Builder 中可视化创建重用窗体

其中 extern "C" 定义是告诉编译器使用 C 命名约定，而不要以 C++ 的命名法，因为 C++ 的命名法会在函数名称后加上参数型态等装饰字，如此会造成其他程序如 VC++，VB 等无法使用的困扰。另外，__stdcall 是用来表示使用的参数传入方法。

__declspec(dllexport)表示声明一个输出函数。这个函数的参数定义采取了 Windows API 函数惯用的定义方式，使用 LPCTSTR、LPTSTR 宏。函数有三个参数，它支持调用者提供窗体标题、提示信息，第三个参数用于返回用户选择目录。该函数的代码如下：

Source　范例 DLL 程序的输出函数

```
//---------------------------------------------------------------------------
bool _stdcall SelectDirectory(LPCTSTR Title, LPCTSTR Msg,LPTSTR Dir)
{
        Form1 = new TForm1(NULL);
        Form1->Caption=AnsiString(Title);
        Form1->Edit1->Text=Form1->DirectoryListBox1->Directory;
        Form1->Label1->Caption=AnsiString(Msg);
        if (Form1->ShowModal()==ID_OK){
              strcpy(Dir,Form1->Edit1->Text.c_str());
              delete Form1;
              return true;
        }
        delete Form1;
        return false;
}
//---------------------------------------------------------------------------
```

完成以上代码之后，就可以编译 DLL 工程。此时 C++ Builder 会产生一个 DLL 文件和 LIB 文件，以本例而言，它会产生一个 SelDir.DLL 文件，而该文件就是供其他应用程序外部调用的动态链结库。

14.1.2 使用 DLL 实现窗体重用

在应用程序中使用一个 DLL 有两种方式：静态导入（Statically Load）和动态导入（Dynamically Load）。本节将讨论如何静态导入并调用 DLL 中的函数，在本书的第 15 章还将深入分析 DLL，讲述如何在应用程序中动态导入 DLL。

一个问题是，如何获得一个未知 DLL 的信息？例如，有某个 DLL，并且知道该 DLL 可以实现某个功能，但如何得知具体的函数调用约定？简单的方法是使用 Windows 95/98 的快速查看工具，它允许用户研究一个 DLL 文件（如图 14-3 所示）。

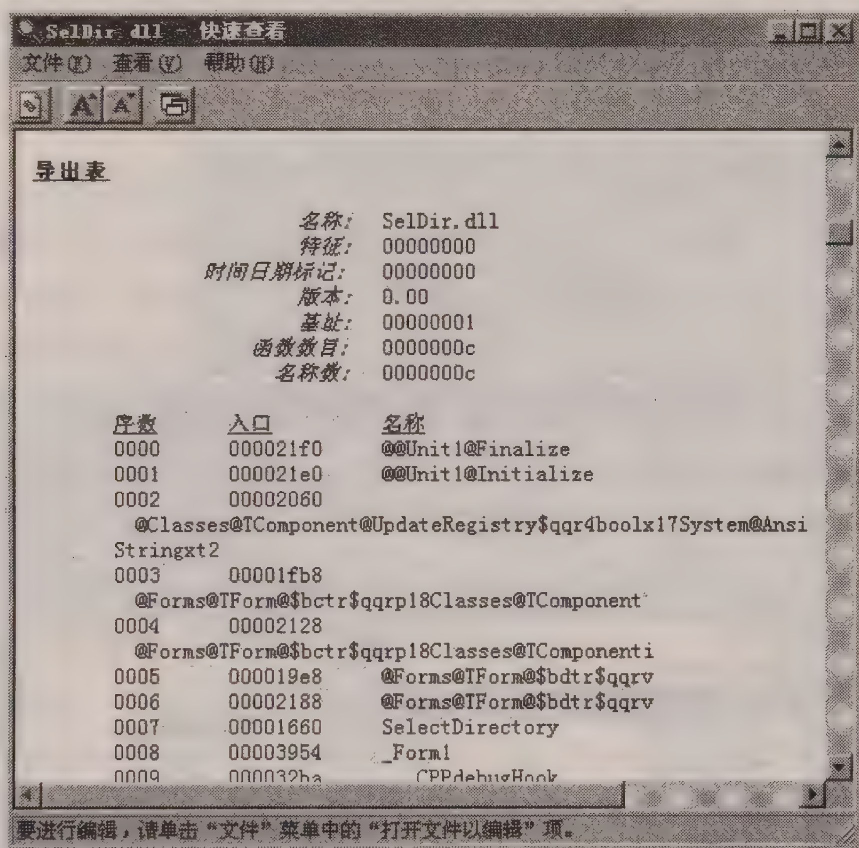


图 14-3 使用快速查看工具查看 SelDir.dll 文件的详细信息

除了快速查看之外，还可以使用一些专门工具来获得更多的细节信息，例如使用 C++ Builder 中的 TDump32 工具。

现在测试上一小节完成的 SelDir.dll 动态链接库，为此需要创建调用程序，该程序可以相当简单，它仅仅在窗体上放入了一个按钮组件。

范例程序采取静态的方式导入 DLL，静态调用方式要求必须有该 DLL 相应的 LIB。在编译 SelDir 工程时已经得到了这个 LIB 文件。选择【**Add To Project**】将这个 LIB 文件加入本工程中。

然后，要在程序中重新声明需要调用的 DLL 中函数，声明的方法与在生成 DLL 时的函数声明基本相同，只是将 dllexport 改为 dllimport，如下：

```
extern "C" __declspec(dllimport)
bool _stdcall SelectDirectory(LPCTSTR Title, LPCTSTR Msg,LPTSTR Dir);
```

现在在本程序中就可以调用 SelectDirectory 函数，从而实现 VCL 窗体的重用。

Source 调用 DLL 中的函数，实现窗体重用

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    char Dir[100];
    if (SelectDirectory("DLLDemo","请您选择一个目录",Dir)){
        MessageBox(Handle,("您选择了"+AnsiString(Dir)).c_str(),
        "DLLDemo",MB_OK);
    }
}
```

图 14-4 显示了这个范例的执行结果。

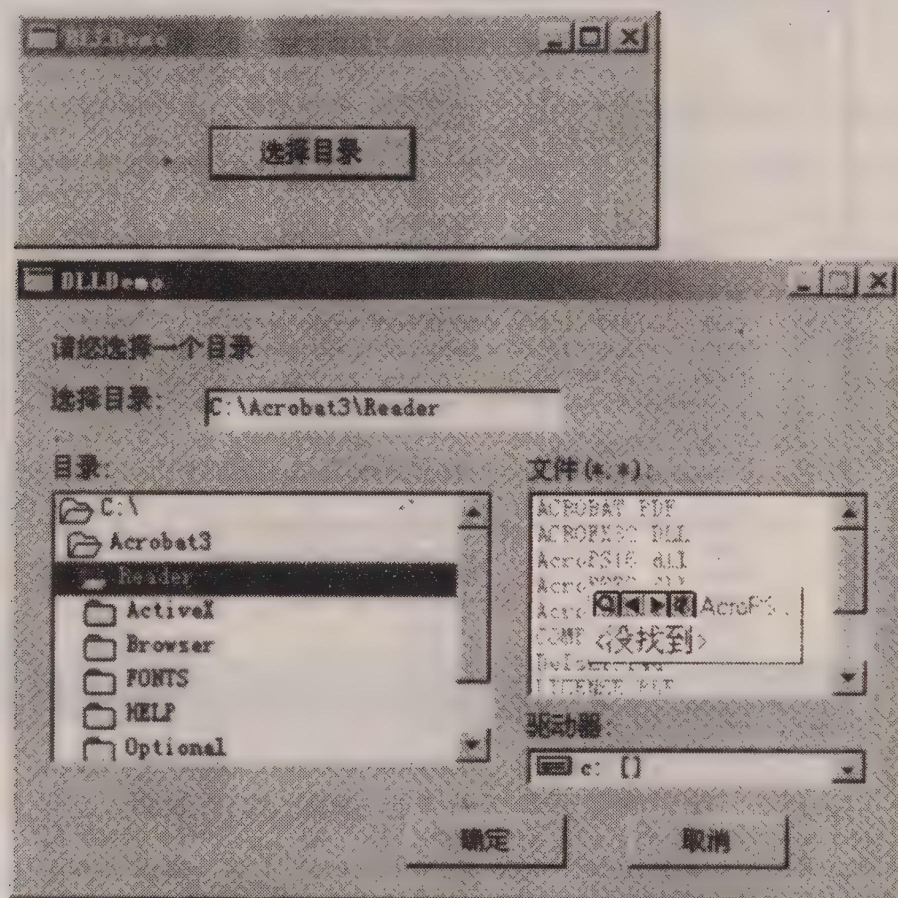
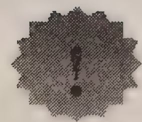


图 14-4 通过 DLL 重用 VCL 窗体



若要使上面的程序能够正确执行，还有一个问题，一定要确保应用程序能够找到 DLL 文件。并非放在硬盘任何地方的 DLL 都可以被使用，只有两种情况下应用程序可以找到并调用 DLL 文件，一是 DLL 和应用程序在同一目录下，例如本例中可以将 SelDir.dll 文件拷到程序的目录下；另外，在环境变量 PATH 域所设定的目录（包括 Windows\、Windows\System\ 等特殊目录）下的 DLL 也可以被找到，例如我们使用 Windows API 函数就属这种情况。

14.2 实例创建分析

本章将创建一个 Windows 系统环境监视程序，这个程序有很丰富的功能，绝大多数是使用 Windows API 函数实现的。所以，本节将首先讲述关于 Windows API 的一些问题。

14.2.1 理解 Windows API

前面提到，Windows API 是基于几个重要的 DLL 的，这些 DLL 可以称之为 Windows 系统 DLL，它们是整个 Windows 系统的核心，其中包括 Windows 必不可少的重要部分 Kernel、User 和 GDI。

在 Windows 中，使用 DLL 是绝对重要的。事实上，DLL 是 Windows 系统的一个关键性的技术基础。因为每一个 Windows 应用程序对于从建立窗口到产生输出的任何一步操作都要和那些系统 DLL 打交道，所以每一个应用程序都必须和那些 DLL 链接。而 Windows API 正是所有的应用程序调用系统 DLL 的接口。

对于 C++ Builder 使用 Windows API 调用系统 DLL 功能来说，不必像上一节中调用自建 DLL 那样麻烦，只需 include 相应的头文件，在程序中就可以随意调用相应的 API 函数。例如，如果需要调用一些系统功能函数，可以加入：

```
#include <winbase.h>
```

而相应的和 DLL 链接的工作编译器会自动完成。

本书曾经多次提到，VCL 的某个组件是封装某一部分 API 的功能，其实，现在所有的开发环境，无论是 Visual C++的 MFC 类库、或者 Delphi、C++ Builder 的 VCL 可视组件库、Borland C++中的 OWL 库，还是 Visual Basic，它们提供创建 Windows 程序的功能本质上还是在调用 Windows API，我们没有直接看到仅仅是因为这些调用被封装了起来。事实上，在当初不存在可视化开发工具时，使用 Windows API 是创建 Windows 程序的唯一选择。

VCL 在效率和易用性方面都足够的优秀，那么是否还有必要再调用 Windows API？答案是肯定的。Windows API 非常庞大，很多功能强大的函数都是 VCL 无法涉及的，其中包括大量重要的系统函数。在前面章节中已经多次验证了上述观点，例如，在第 3 章，我们接触了强大的 ShellAPI；第 8 章，使用 API 函数实现了游戏手柄的支持；第 9 章，我们使用 API 函数创建进程，这样的例子不胜枚举。使用 Windows API 的另外一个非常重要的理由是，C++ Builder 程序可以使用 Windows API 编程中常用的消息处理机制，使用和控制消息可以使程序具有许多意想不到的功能。

如果仅仅使用组件或控件编程一定会像 Visual Basic 和 PowerBuilder 那样只在某一特定领域才有价值，VCL 加 Windows API 的组合才是 C++ Builder 进行 Windows 编程的真谛。特别是对一个 C/C++程序员来说，熟悉 Windows API 是非常必要的。本章将实现一个系统环境监视程序，通过该程序的创建来讲述 Windows API 函数调用的方法和特殊技巧。

14.2.2 程序分析

范例程序——系统环境监视程序的结构较为简单，实际上，系统环境监视程序包含“窗口”、“磁盘驱动器”、“内存资源”、“设备”、“环境变量”、“正在运行程序”和“启动程序”七个模块，并且将这七个功能模块组合在了一起。

程序的主窗体以一个多页面组件 TPageControl 为核心，划分了七个页面分别对应于上述七项功能。同时窗体中包含了一个工具栏和一个状态栏。对于在 TPageControl 组件的每一个 TabSheet 件切换时（即功能模块切换时），程序处理了 OnChange 事件，根据不同的功能模块调整工具栏按钮（使对该功能无效的按钮变灰），并在状态栏输出提示信息。

界面上竖过来的标题栏很有特色（如图 14-5 所示），这个功能也是借助了强大的 Windows API 得以实现。稍后将介绍具体的实现方法。

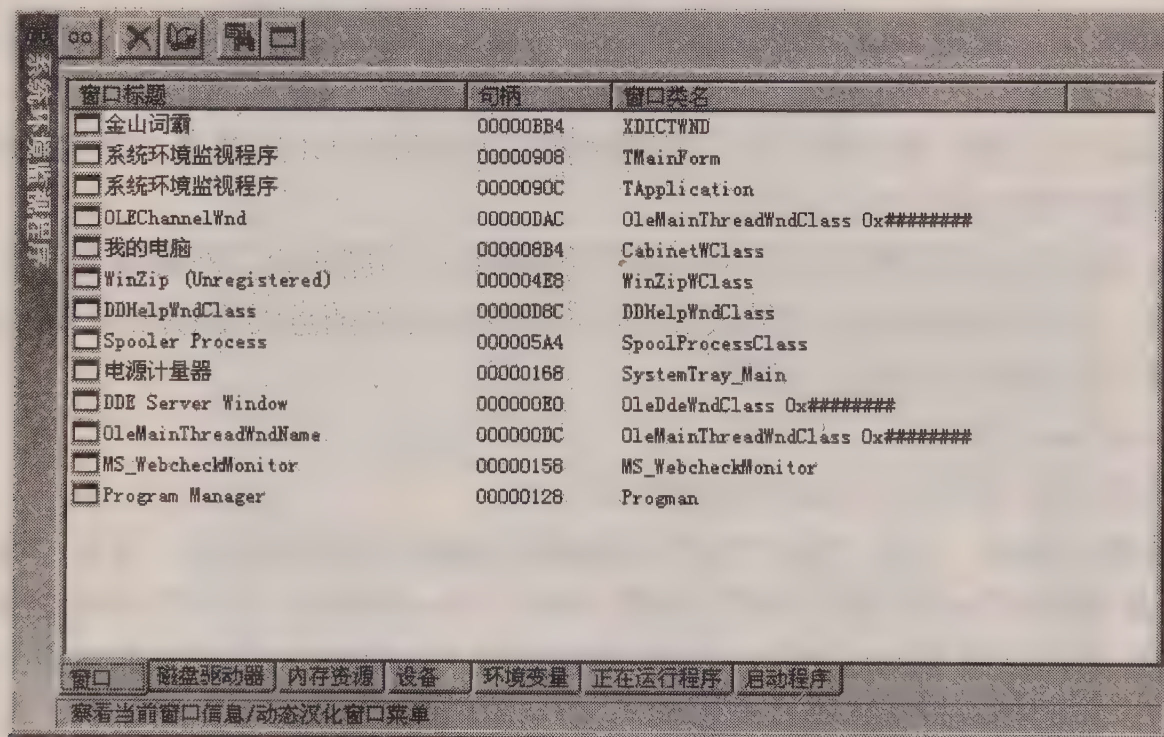


图 14-5 系统环境监视程序的一页

系统环境监视程序是具备任务栏指示区图标的程序。这种程序是 Win32 的新特色，目前，Windows 95/98 中这种“任务栏图标”应用程序已经随处可见，这种效果也只能通过 Windows API 来实现。

系统环境监视程序的功能很多，其中不乏一些很有特点的功能，例如动态汉化菜单、杀除进程等等，如果不知道相应的 API 函数的话，实现这些功能简直是无法想象的。通过本章读者将可以了解到，使用 Windows API 这些颇具技巧性的编程任务也并非很难。

在实现本章范例程序的过程中，将会讨论几个 Windows API 编程的重要概念，包括消息机制和处理消息，自定义消息，回调函数等等。

SystemView 工程中包含下面的一些文件：

- 主窗体单元。文件：Unit1.cpp、Unit1.h。
- 动态汉化菜单所需的字典文件. 文件：Eng.txt、Chs.txt。

14.3 编写任务栏指示区图标支持

在 Windows 95/98/NT 的桌面上，位于屏幕的边缘（默认在下方），有一个特殊的窗口称为任务栏。在任务栏的一端有一块区域称之为任务指示区。在 Win32 系统中，一些应用程序可以只作为指示图标运行，即在任务指示区显示一个图标来表示应用程序或工具的运行状态。

很多著名的软件，如金山词霸、Norton AntiVirus 都会在任务栏指示区加入自己的图标。对于本章要完成的监控程序，做成“任务栏图标程序”是相当适合的。

创建任务栏指示图标程序只涉及了一个 API 函数——Shell_NotifyIcon。使用此函数需要包含 shellapi.h 头文件。

这个函数有两个参数，第一个参数为是否要添加、删除、改动图标的标志，第二个参数指向一个 NOTIFYICONDATA 结构。下面是函数 Shell_NotifyIcon 的声明：

```
WINSHELLAPI BOOL WINAPI Shell_NotifyIcon(
    DWORD dwMessage, // message identifier
    PNOTIFYICONDATA pnid   // pointer to structure
  );
```

参数 dwMessage 是发送消息的标识，有下面三个可取值：

NIM_ADD	向任务栏指示区加入图标。
NIM_DELETE	删除任务栏指示区的图标。
NIM_MODIFY	改变指示图标。

第二参数指向一个较为复杂的 NOTIFYICONDATA 结构，该结构形式如下：

```
typedef struct _NOTIFYICONDATA { // nid
    DWORD cbSize;
    HWND hWnd;
    UINT uID;
    UINT uFlags;
    UINT uCallbackMessage;
    HICON hIcon;
    char szTip[64];
} NOTIFYICONDATA, *PNOTIFYICONDATA;
```

下面是 NOTIFYICONDATA 结构各个域的说明：

- cbSize 是该结构的大小（系统用来确定版本），设为 sizeof (NOTIFYICONDATA)。
- hWnd 是一个窗口句柄，指示图标会向这个窗口发送消息。
- uID 是指示区显示图标的 ID 标识，只有当一个应用程序有多个指示图标是才有用。
- uFlags 说明下面三个域是否有效，可取值为 NIF_ICON、NIF_MESSAGE、

NIF_TIP。

- uCallbackMessage 为回调消息，该消息会被发送到 hWnd 指向的窗口，通知窗口用户对图标进行某种操作。
- hIcon 指明在任务栏指示区显示的图标。
- szTip 提供一段提示文本，当鼠标移动到任务栏图标上时，显示这段文本。

使用 Shell_NotifyIcon 函数具有一些特殊性，关键在于我们需要自定义一个回调消息，并定义一个消息处理函数。在主窗体单元头文件中定义消息：

```
#define WM_NOTIFYICON      WM_USER+5
```

用户自定义消息应该从 WM_USER 宏开始，因为在 WM_USER 宏之前的消息号已经为系统消息所用。

然后进行消息映射，可以看到除了 WM_NOTIFYICON 消息外，还应该映射 WM_NCHITTEST 系统消息，此消息用于实现特殊风格标题栏的功能。

```
BEGIN_MESSAGE_MAP
 MESSAGE_HANDLER(WM_NOTIFYICON,TMessage,NotifyIcon)
 MESSAGE_HANDLER(WM_NCHITTEST,TMessage,NCHitTest)
END_MESSAGE_MAP(TForm)
```

之后，需要编写 NotifyIcon 函数处理回调消息，在该函数中，我们对用户在图标上按下鼠标左键和鼠标右键进行了处理。

Source 编写函数处理 WM_NOTIFYICON 回调消息

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::NotifyIcon(TMessage& Msg)
{
        POINT MousePos;
        switch(Msg.LParam)
        {
        case WM_RBUTTONUP:
                if (GetCursorPos(&MousePos))
                {
                        PopupMenu1->PopupComponent = MainForm;
                        SetForegroundWindow(Handle);
                        PopupMenu1->Popup(MousePos.x, MousePos.y);
                }
                break;
          case WM_LBUTTONDOWN:
                Show();
                break;
          default:
                break;
```

```
        }
        TForm::Dispatch(&Msg);
}
//---------------------------------------------------------------------------
```

在这里使用了 API 函数 GetCursorPos 来获取用户按下鼠标右键时鼠标的位置，然后将一个在 C++ Builder 中做好的 PopupMenu 在鼠标所在位置显示出来。

如果用户按下鼠标左键，则将主窗体显示出来。

在完成自定义回调消息和处理消息的函数后，就可以调用 Shell_NotifyIcon 函数来创建一个任务栏指示区图标了。本程序将建立一个 TrayMessage 函数来处理 Shell_NotifyIcon 函数的调用。代码如下：

Source 编写函数处理 Shell_NotifyIcon 函数调用

```
//---------------------------------------------------------------------------
bool __fastcall TMainForm::TrayMessage(DWORD dwMessage)
{
        NOTIFYICONDATA tnd;
        tnd.cbSize  = sizeof(NOTIFYICONDATA);
        tnd.hWnd     = Handle;
        tnd.uFlags  = NIF_MESSAGE | NIF_ICON | NIF_TIP;
        tnd.uCallbackMessage     = WM_NOTIFYICON;
        tnd.hIcon=myIcon->Handle;
        lstrcpyn(tnd.szTip,"系统环境监视程序",sizeof(tnd.szTip));
        return (Shell_NotifyIcon(dwMessage, &tnd));
}
//---------------------------------------------------------------------------
```

最后，可以在 FormCreate 函数中加入 TrayMessage(NIM_ADD)创建一个任务栏指示区图标，在 FormCreate 函数调用 TrayMessage(NIM_DELETE)消除指示区图标。

还有一个问题，是否可以像“金山词霸”那样在程序启动时仅仅加入一个任务栏指示区图标，却不显示程序的主窗体？这并不难以办到。打开主入口 CPP 文件，在本例中是 SystemView.cpp，在 Application->CreateForm 语句之前加入如下两行：

```
ShowWindow(Application->Handle,SW_HIDE);
Application->ShowMainForm=false;
```

对于任务栏指示图标程序，需要提供主窗体的隐藏功能，是主窗体不出现而只保留一个任务指示区的图标。这只需使用 TForm 类的 Hide 方法即可，如下：

```
void __fastcall TMainForm::ToolButton6Click(TObject *Sender)
{
        Hide();
}
```

程序运行的结果如图 14-6 所示。

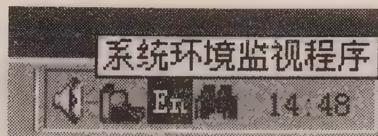


图 14-6　向任务栏指示区添加图标

14.4　利用 API 实现特殊风格的标题栏

一般来说，Windows 应用程序都有风格一致的界面，虽然，Microsoft 为 Windows 应用程序定做的界面是很考究的，但千篇一律的程序界面总会让人有感到厌烦的时候。在本章的系统环境监视程序中，我们想改变一下这种状况，系统环境监视程序将让窗体的标题栏从一般的上边移动到窗体的左边，并且这个标题栏还具备原有标题栏的功能。

14.4.1　自绘标题栏

事实上，在 Windows 程序中，标题栏作为窗口的一部分与窗口密不可分，想要使标题栏从上方“移动”到左侧是没有可能的，虽然窗口的风格是较为固定的，但编程却可以非常灵活。采用这样的步骤解决这个问题：

(1)　去掉窗体的原有标题栏。

(2)　在窗体的左侧绘制一个标题栏。

(3)　使自绘标题栏具有标准的标题栏功能。

图 14-7 给出了设计期间的系统环境监视程序。

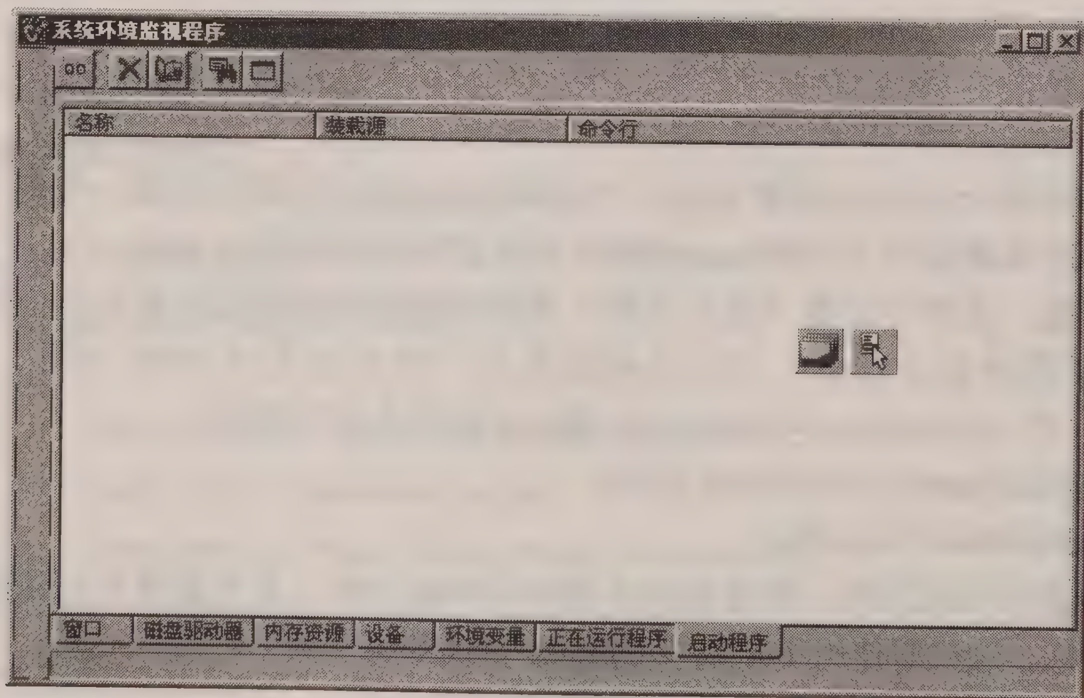


图 14-7　设计期间的系统环境监视程序

如何去掉在窗体上方的标题栏？读者可能会想到将 Form 的 BorderStyle 属性设置为 bsNone，这样做确实可以去掉标题栏，但是，它也同时去掉了窗体的边框，这会非常不美观，有违初衷。是否可以只去掉窗体的标题栏而不影响边框？可以通过重载 CreateParams 函数解

决这个问题。

以下是重载的 CreateParams 函数。

Source 重载 CreateParams 函数改变窗体样式

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::CreateParams(TCreateParams& Params)
{
        TForm::CreateParams(Params);
        Params.Style ^= WS_DLGFRAME;
        Params.Style |= WS_POPUP;
}
//---------------------------------------------------------------------------
```

CreateParams 是一个虚函数，可以经由重载它来修改 Windows 的 Style，因为原先在 C++ Builder 中所定义的 Form 是一个 WS_DLGFRAME 风格，而这种对话框框架的外形内定是有标题栏的。

但是在 Windows 系统中除了前面的 WS_DLGFRAME 风格，还提供了另一种 WS_POPUP 风格的窗口，只是在 C++ Builder 并未提供这种选项。所以只能通过改写 CreateParams 的方式来产生 WS_POPUP 形式的窗口，只需将 Params.Style 的 WS_DLGFRAME （代表使用对话框外框）改成另一种 WS_POPUP （弹出式外框），这只要利用按位或及按位与运算就可以达到。

其后的工作是如何绘制一个具有 Windows98 风格的标题栏。首先，为了使绘制标题栏不会影响到其他窗体中的元素，需要使用一个 TPaintBox 组件和一个 TPanel 组件分割窗体。左边的 TPaintBox 组件设置为要绘制标题栏的宽度，Align 属性设为 alLeft，右边的 Panel 的 Align 属性设为 alClient，这个 Panel 将成为实际上的窗口区域（如图 14-7 所示）。

下面着手在 PaintBox 上绘制标题栏，为了绘制具有 Windows 98 渐变颜色的标题栏，程序中在头文件中定义了标题栏的起始颜色和终止颜色，如下：

```
const int sR=255,sG=0,sB=0;          // 纯红
const int eR=255,eG=255,eB=0;        // 纯黄
```

响应主窗体的 OnPaint 事件，绘制标题栏的全部，函数如下：

Source FormPaint 函数，绘制标题栏

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::FormPaint(TObject *Sender)
{
        Graphics::TBitmap *tmpBitmap=new Graphics::TBitmap();
        tmpBitmap->Width=PaintBox1->Width;
        tmpBitmap->Height=PaintBox1->Height;
        int n=tmpBitmap->Width/5+tmpBitmap->Height/5;
```

```
    int     RInc=(eR-sR)/n,
           GInc=(eG-sG)/n,
            BInc=(eB-sB)/n;
    TColor FillColor;
    TRect rect;
    // 绘制渐变色标题栏
    for (int i=0;i<tmpBitmap->Width;i+=5){
    for (int j=0;j<tmpBitmap->Height;j+=5){
            FillColor=RGB(sR+RInc*(i/5+j/5),
            sG+GInc*(i/5+j/5),sB+BInc*(i/5+j/5));
            tmpBitmap->Canvas->Brush->Color=FillColor;
            tmpBitmap->Canvas->FillRect(
                    TRect(i,j,i+5,j+5));
    }
    }
    // 绘制标题栏图标
    ImageList1->Draw(tmpBitmap->Canvas,4,4,5,true);
    // 绘制窗口标题
    tmpBitmap->Canvas->Brush->Style=bsClear;
    char* msg="系统环境监视程序";
    LOGFONT fontRec;
    memset(&fontRec,0,sizeof(LOGFONT));
    fontRec.lfHeight = -14;
    fontRec.lfWeight = FW_BOLD;
    fontRec.lfEscapement =2700; // 旋转文字 270 度
    lstrcpy(fontRec.lfFaceName,"宋体");
    HFONT hFont=CreateFontIndirect(&fontRec);
    ::SelectObject(tmpBitmap->Canvas->Handle,hFont);
    ::SetTextColor(tmpBitmap->Canvas->Handle,RGB(255,255,255));
    ::TextOut(tmpBitmap->Canvas->Handle,18,22,msg,lstrlen(msg));
    ::DeleteObject(hFont);
    PaintBox1->Canvas->Draw(0,0,tmpBitmap);
    delete tmpBitmap;
}
//---------------------------------------------------------------------------
```

这个重绘函数完成三件事：绘制标题栏渐变底色；绘制图标；绘制窗口标题。对于绘制渐变颜色，程序是将标题栏区域分成数个 5×5 像素的区域，然后算出每个区域在渐变过程的颜色，并填充每一个区域。这样结果就画出了漂亮的从红到黄的渐变颜色了。

图标绘制较为容易，在程序中直接将 ImageList 组件中的一个图标通过 Draw 方法在标题栏上部绘制出来。

绘制窗口标题是一个较为棘手的问题，这里不能简单的使用 Canvas 类的 TextOut 方法输出文字，因为需要“竖过来” 绘制文本，也就是要将文件旋转 270 度，这种需求 VCL 的图形类已经不能满足了，必须使用 Windows API 中的 GDI 函数。

可以看到，程序中使用 GDI 函数建立了一个字体，使用 API 建立字体需要一个 LOGFONT 结构，该结构的 lfEscapement 域允许字体旋转功能。另外，C++ Builder 中 TCanvas 类的 Handle 属性就相当于 API 中 HDC，通过该属性可以与 GDI 函数协同工作。

14.4.2 实现标准标题栏功能

以上已经将位于窗口左侧的标题栏绘制完成，但现在它还只能是一个摆设，并不具备真正的窗口标题的功能，如何能够使自绘的标题栏更像一个真正的标题栏呢?

这需要利用一个特殊的 Windows 消息——WM_NCHITTEST。WM_NCHITTEST 消息是用来决定目前鼠标所在位置属性的消息，因此可以利用此特性，当鼠标移至指定的位置时，传回 HTCAPTION，使得系统以为鼠标目前位于标题栏，如此程序中绘制的标题栏就可以实现标准 Windows 标题栏的功能了，例如通过标题栏窗体拖动，双击标题栏最大化窗口等。

首先还是进行消息映射工作，具体来说就是设定 NCHitTest 函数处理 WM_NCHITTEST 消息。该函数的具体内容如下：

Source 编写处理 WM_NCHITTEST 消息的函数

```
void __fastcall TMainForm::NCHitTest(TMessage& Msg)
{
        TPoint pt;
        pt.x=LOWORD(Msg.LParam);
        pt.y=HIWORD(Msg.LParam);
        pt =ScreenToClient(pt);
        RECT rc;
        ::SetRect(&rc,0,0,20,ClientHeight);
        if (PtInRect(&rc,pt))
            Msg.Result= HTCAPTION;
        else
            Msg.Result= HTCLIENT;
}
```

这段代码容易理解，首先，从消息中分解出当前鼠标的位置信息，然后，使用 TForm 类的 ScreenToClient 方法将鼠标位置（屏幕坐标）转换为在窗体中的坐标，再后，判断目前鼠标位置是否位于自绘的标题栏区域内，如果是，则告诉系统，用户单击了标题栏，否则告诉系统单击了窗体客户区。

上述代码会成功地欺骗系统，这样，范例程序中自绘的标题栏从内到外就具备了标准标题栏的一切特性了。

14.5　窗口与程序

本节将实现范例程序中窗口和程序监控部分的功能，这涉及到范例程序中的三个模块，“窗口”、“正在运行程序”和“启动程序”。

14.5.1　获得当前所有窗口

获得当前所有窗口可以使用 Windows API 函数 EnumWindows。该函数可以枚举当前所有顶级窗口。这个函数声明如下：

```
BOOL EnumWindows(
    WNDENUMPROC lpEnumFunc, // 回调函数
    LPARAM lParam
  );
```

这个API函数比较典型，它使用了一个函数指针作为参数，这种情况在普通的C++ Builder编程中是不常遇到的，而在 Windows API 调用中却比较常见。EnumWindows 函数的第一个参数指向一个回调函数，调用 EnumWindows 后，每查到一个顶级窗体，就会调用这个回调函数，直到查完所有的顶级窗体。第二个参数是传给回调函数的值，在本程序中，此参数无用。

以下编写 EnumWindows 的回调函数，WNDENUMPROC 宏所定义的回调函数为下面的形式：

```
BOOL CALLBACK EnumWindowsProc(
    HWND hwnd,          // 父窗体句柄
    LPARAM lParam       // 程序传递值
  );
```

所以在创建回调函数时，必须符合这样的形式，编写回调函数如下：

Source　EnumWindows 的回调函数 EnumProc

```
//---------------------------------------------------------------------------
bool __stdcall EnumProc(HWND hWnd, LPARAM lp)
{
        if(hWnd==NULL)
            return false;
        char title[60];
        char hwndStr[10];
        char className[60];
        TListItem *mItem;
```

```
            GetWindowText(hWnd,title,60);
            if (AnsiString(title)!=EmptyStr&&AnsiString(title)!="Default IME"){
                mItem=MainForm->ListView1->Items->Add();
                sprintf(hwndStr,"%08x",hWnd);
                mItem->Caption=AnsiString(title);
                mItem->ImageIndex=4;
                mItem->SubItems->Add(AnsiString(hwndStr).UpperCase());
                GetClassName(hWnd,className,60);
                mItem->SubItems->Add(AnsiString(className));
            }
            return true;
    }
    //---------------------------------------------------------------------------
```

回调函数通过参数获得了找到的窗口句柄，通过该窗口句柄，可以调用 GetWindowText 和 GetClassName 两个 API 函数获得该窗口标题和窗口类名，这两个函数的调用方式非常简单，相信读者一看便知。最后，将窗口句柄、窗口标题和窗口类名加入一个 TListView 列表视图，以显示出来，如图 14-8 所示。

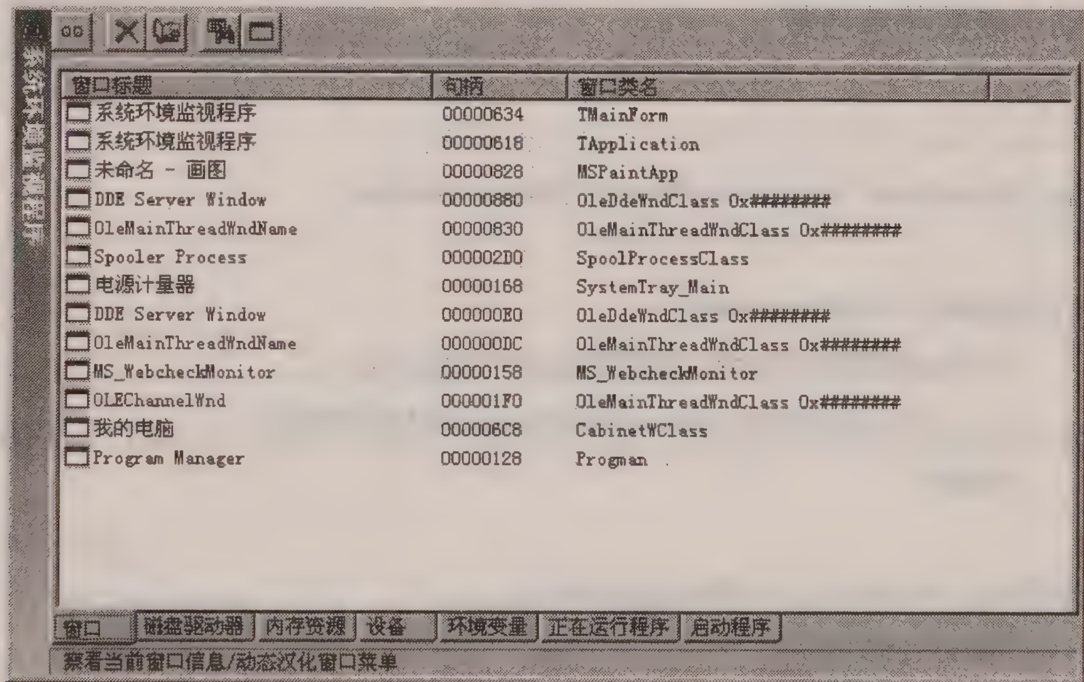


图 14-8　显示当前 Windows 中的所有窗口的信息

现在，在程序中就可以调用 EnumWindows 函数，注意调用时要将回调函数指针强制转换为 WNDENUMPROC，如下：

```
EnumWindows((WNDENUMPROC)EnumProc,0);
```

使用 EnumWindows 是一种最为简单的获得所有窗口的方法，其实，还可以使用 Windows API 函数 GetDesktopWindow 和 GetWindow 来手工枚举所有顶级窗口。

另外，可以仅仅显示顶级可视窗口，这样列出的窗口就不会繁杂无序，而仅会列出在任务栏上出现的窗口。实现此功能需要使用 API 函数 IsWindowVisible 判断一下，可以这样写：

```
if (IsWindowVisible(hWnd)){
    ......
```

14.5.2　动态汉化窗口菜单

本节将讲解如何实现动态汉化菜单的有趣功能。动态汉化功能包含在范例程序的“窗口”模块中，用户可以在列表视图中选择一个窗口进行汉化。

上一小节已经看到如何获得一个窗口的句柄，而有了窗口句柄，就有了控制该窗口的钥匙，通过窗口句柄，可以进一步获得窗口的菜单，进而又可以调用相应的 Windows API 函数修改菜单内容。

具体来说，实现动态汉化菜单功能应遵循以下步骤：

(1)　通过窗口句柄获得窗口菜单句柄。

(2)　获得顶行菜单数目，通过菜单位置序号遍历顶行菜单。

(3)　获得顶行菜单的标题，查字典，修改菜单标题。

(4)　按上述方法继续遍历下拉菜单和二级菜单，修改菜单。

这样说可能还不清楚，请看下面的具体代码。

Source　动态函数菜单函数

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ToolButton3Click(TObject *Sender)
{
        if (ListView1->SelCount==0){
            MessageBox(Handle,"请首先选择一个窗口项！",
                "汉化程序菜单",MB_OK|MB_ICONWARNING);
            return;
        }
        HMENU childMenu,mainMenu,c2Menu;
        char menuStr[30];
        int mainNum,childNum,c2Num;
        AnsiString newStr;
        int a,b,c;
        HWND hWnd=FindWindow(
        ListView1->Selected->SubItems->Strings[1].c_str(),
        ListView1->Selected->Caption.c_str()); // 获得窗口句柄
        if (hWnd!=NULL){
            mainMenu=GetMenu(hWnd); // 获取窗口菜单句柄
            mainNum=GetMenuItemCount(mainMenu); // 获取顶级菜单个数
            for (a=0;a<mainNum;a++){
```

```
            GetMenuString(mainMenu,a,menuStr,30,MF_BYPOSITION); // 获取菜单标题
            if ((newStr=Convert(menuStr))!="")  // 查字典
                ModifyMenu(mainMenu,a,MF_BYPOSITION,
                GetMenuItemID(mainMenu,a),newStr.c_str()); // 修改菜单
            childMenu=GetSubMenu(mainMenu,a);// 获取该菜单的下拉菜单（子菜单）句柄
            childNum=GetMenuItemCount(childMenu); // 子菜单菜单项个数
            for (b=0;b<childNum;b++){ // 按同样方式遍历并修改菜单
                GetMenuString(childMenu,b,menuStr,30,MF_BYPOSITION);
                if ((newStr=Convert(menuStr))!="")
                    ModifyMenu(childMenu,b,MF_BYPOSITION,
                GetMenuItemID(childMenu,b),newStr.c_str());
                c2Menu=GetSubMenu(childMenu,b); // 获取二级菜单句柄
                c2Num=GetMenuItemCount(c2Menu); // 获取二级菜单菜单项个数
                for (c=0;c<c2Num;c++){
                    GetMenuString(c2Menu,c,menuStr,30,MF_BYPOSITION);
                    if ((newStr=Convert(menuStr))!="")
                        ModifyMenu(c2Menu,c,MF_BYPOSITION,
                    GetMenuItemID(c2Menu,c),
                    newStr.c_str());
                }
            }
        }
    }
}
//------------------------------------------------------------------------
```

这里首先使用了 FindWindow 函数获得要汉化菜单的窗口句柄。FindWindow 函数声明如下：

```
HWND FindWindow(
    LPCTSTR lpClassName,         // 窗体类名
    LPCTSTR lpWindowName         // 窗口标题
  );
```

FindWindow 函数的两个参数分别表示要寻找的窗口的标题和窗口类，而窗口标题和窗口类名在“获得当前所有窗口”功能的实现过程中已经得到。

具体汉化菜单用到了这样一些和菜单相关的 Windows API 函数，包括：GetMenu、GetMenuItemCount、GetSubMenu、GetMenuString 和 ModifyMenu。前三个函数很简单，此处不再赘述。ModifyMenu 函数用于修改一个菜单项，函数声明如下：

```
BOOL ModifyMenu(
  HMENU hMnu,      // 菜单句柄
  UINT uPosition,  // 调整菜单项位置
  UINT uFlags,     // 菜单项标志
```

```
    UINT uIDNewItem,        // 菜单项 ID
    LPCTSTR lpNewItem    // 内容
    );
```

ModifyMenu 函数用来修改某一菜单项的标题，或者该菜单项的子菜单。uFlags 参数有两个可取值：MF_BYCOMMAND 和 MF_BYPOSITION，表示根据标识还是根据位置调整菜单。

这段程序中调用了 Convert 函数，这个函数用于生成菜单的替换标题，即汉化菜单。本程序在 FormCreate 函数中，建立了两个 TStringList 类的实例，分别使用 LoadFromFile 方法装入一组英文单词和相应的一组汉语解释。

```
eList=new TStringList();
cList=new TStringList();
eList->LoadFromFile("Eng.txt");
cList->LoadFromFile("Chs.txt");
```

在 Convert 函数中，程序分析了菜单的原标题，分解出标题中的快捷键（如&A）和加速键(如 Ctrl+A)，并对菜单的“正文”部分查字典，对于查到的单词进行替换，最后，重新组合该快捷键、加速键和“正文”，生成新的菜单标题。

Source 转换英文的菜单标题

```
AnsiString __fastcall TMainForm::Convert(char * menuStr)
{
        AnsiString Str1=AnsiString(menuStr);
        AnsiString Str2="";
        int pos=Str1.Pos("&");
        if (pos!=0)
            Str2="(&"+AnsiString(Str1[pos+1])+")";
        Str1.Delete(pos,1);
        Str1=Str1.UpperCase();
        for (int i=0;i<eList->Count;i++){
            pos=Str1.Pos(eList->Strings[i]);
            if (pos!=0){
                Str1.Delete(pos,eList->Strings[i].Length());
                Str1.Insert(cList->Strings[i],pos);
                return Str2.UpperCase()+Str1;
            }
        }
        return EmptyStr;
}
```

图 14-9 显示了使用系统环境监视程序动态汉化 ACDSee32 英文版菜单的结果。

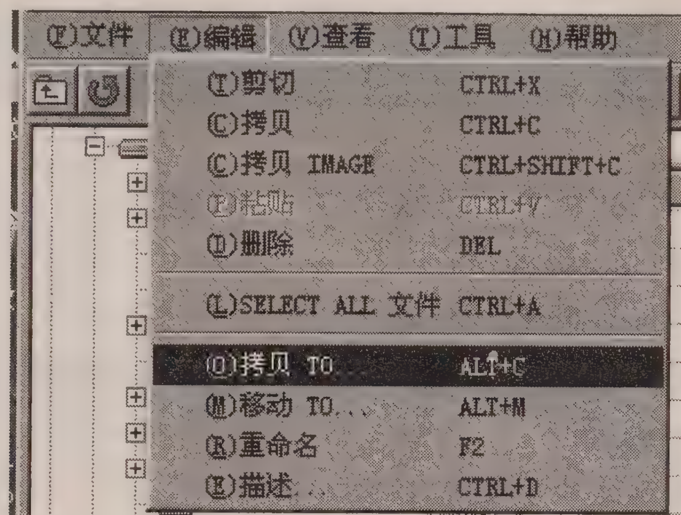


图 14-9　动态汉化菜单的效果

14.5.3　获得当前激活的进程

最为简便的获得当前激活的进程信息的方法是使用 ToolHelp API 函数，ToolHelp 函数库可以使程序员更为容易地获得现在执行的 Win32 应用程序的信息。使用这组函数必须包含 tlhelp32.h 头文件。

使用 ToolHelp API 首先应该调用 CreateToolhelp32Snapshot 函数，该函数可以产生一个当前系统中所有进程、堆、模块以及线程信息的“快照”。这个函数的声明如下：

```
HANDLE CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID);
```

参数 dwFlags 指定获取哪一类信息。经常使用的是以下几个值：

TH32CS_SNAPHEAPLIST	在“快照”中包含指定进程的堆的列表
TH32CS_SNAPMODULE	在“快照”中包含指定进程的模块的列表
TH32CS_SNAPPROCESS	在“快照”中包含当前激活的所有进程
TH32CS_SNAPTHREAD	在“快照”中包含当前所有线程的列表

参数 th32ProcessID 指定一个进程。如果取 0 则表示当前的所有进程。该参数仅在 dwFlags 为 TH32CS_SNAPHEAPLIST 和 TH32CS_SNAPMODULE 时有效。

在成功的获得系统快照后，应该使用一组 First/Next 函数获取系统快照中的详细信息。比如本例使用 Process32First 和 Process32Next 函数获取进程信息，类似的函数还有 Module32First、Module32Next、Thread32First、Thread32Next 等。释放系统快照，可以使用 CloseHandle API 函数。

下面是实例程序中获取所有进程信息的具体代码。演示了 ToolHelp API 函数和获取程序版本信息的 Windows API 函数的使用。

Source 获得当前激活进程

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ViewProcess()
{
    ListView2->Items->Clear();
    TListItem *mItem;
```

```
        AnsiString ExeFile;
        Pointer pt,pt2;
        unsigned int s;
        DWORD size,size2;
        HANDLE snapshot;
        PROCESSENTRY32 processinfo;
        processinfo.dwSize = sizeof(processinfo);
        snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
        if (snapshot==NULL) return;
        bool flag = Process32First (snapshot,&processinfo);
        while (flag){
            mItem=ListView2->Items->Add();
            ExeFile=AnsiString(processinfo.szExeFile);
            mItem->Caption=ExeFile;
            mItem->SubItems->Add(
            IntToStr(int(processinfo.th32ParentProcessID)));
            mItem->SubItems->Add(
            IntToHex(int(processinfo.th32ProcessID),8).UpperCase());
            size=GetFileVersionInfoSize(ExeFile.c_str(),&size2);
            pt=malloc(size);
            GetFileVersionInfo(ExeFile.c_str(),NULL,size,pt);
            if(VerQueryValue(pt,"\\StringFileInfo\\040904E4\\FileVersion",&pt2,&s))
                mItem->SubItems->Add(PChar(pt2));
            if(VerQueryValue(pt,"\\StringFileInfo\\040904E4\\CompanyName",&pt2,&s))
                mItem->SubItems->Add(PChar(pt2));
            if(VerQueryValue(pt,"\\StringFileInfo\\040904E4\\FileDescription",&pt2,&s))
                mItem->SubItems->Add(PChar(pt2));
            free(pt);
            flag = Process32Next(snapshot,&processinfo);
        }
        CloseHandle(snapshot);
    }
    //---------------------------------------------------------------------------
```

可以看到，这段程序在一个 while 循环中遍历了所有的进程。遍历系统快照需要调用 Process32First 和 Process32Next 函数，这两个函数获得进程信息并返回到一个 PROCESSENTRY32 的结构中，该结构包含一个 th32ProcessID 域可以获得进程的 ID，th32ParentProcessID 域可以获得该进程父进程的 ID。szExeFile 域对应该进程的可执行文件的文件名。

当然我们并不满足仅仅获得上述信息。在实例程序中，进一步调用一组获得版本信息的函数，由此可以得到该进程的更多内容，包括：版本号、公司名和程序描述，如图 14-10 所示。

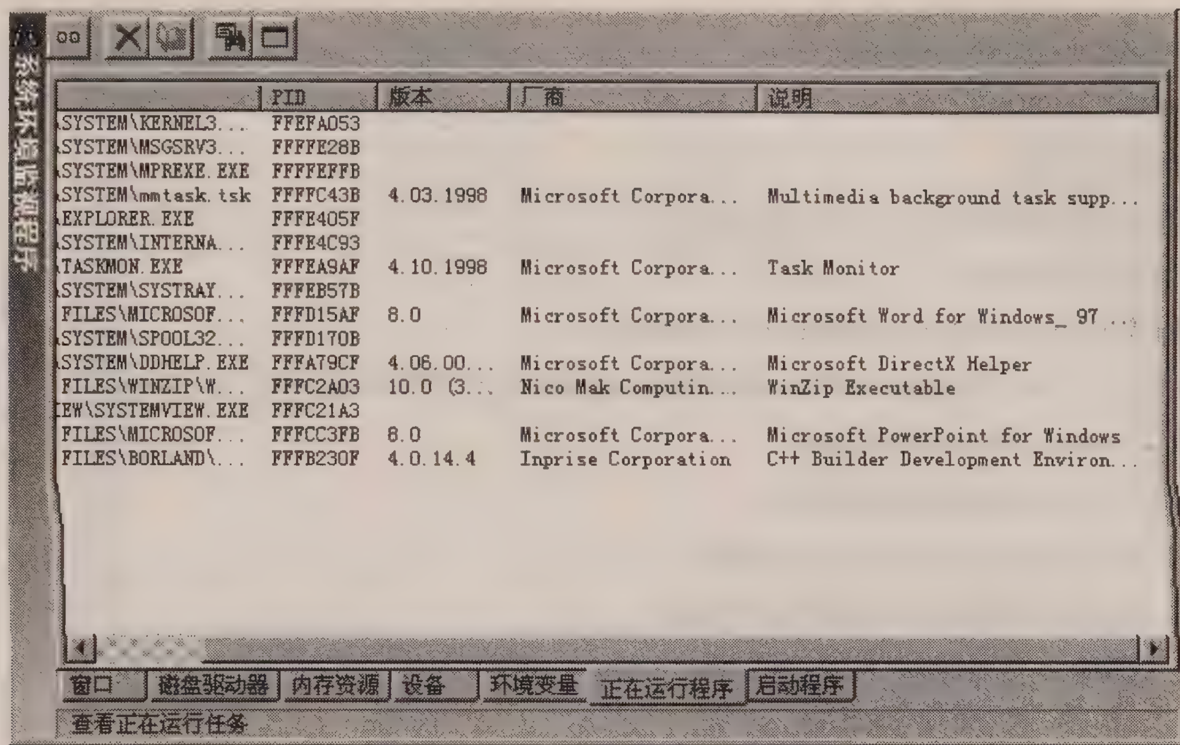


图 14-10 查看当前的所有进程信息

获取程序版本信息功能的实现并不像想像中那样简单。其核心的 API 调用是 GetFileVersionInfo。该函数需要文件名，指向内存存储数据区域的指针，以及该内存段的大小作为参数。为了获得合适的大小，应该调用 GetFileVersionInfoSize 函数，调用这个函数一般是必要的，因为需要的内存区域往往很大。获得具体的版本信息可以调用 VerQueryValue 函数，该函数的第一个参数指向信息数据的指针，第二个参数为一个指示所需信息的位置的字符串，第三、第四个参数指明返回信息的指针，和返回内容的大小。

14.5.4 查看/删除系统启动程序

在 Windows 98/95 系统启动时，会同时启动一些程序。这些启动程序的信息被填写在系统注册表中，可以通过读取和修改注册表来查看/删除系统启动程序。

这项功能是很实用的，因为某些商业软件会自动在注册表中加入一些启动程序，而想要使这些程序不启动必须手工修改注册表，这是非常不便的。本程序可以在这个模块中方便的管理系统启动程序。

这部分功能的实现并没有使用 API 函数，而是使用了 C++ Builder 的 TRegistry 类来操作注册表，使用 TRegistry 类操作注册表的方法在本书的第 6 章中已经有所讲述。

所以，实现这部分功能仅仅需要对 Windows 注册表进行了解。实际上，系统启动程序的信息存在于注册表的两个地方，分别是 Windows 主键下的 CurrentVersion\ RunServices 和 CurrentVersion\Run。下面是实现这一功能的具体代码。

Source 查看系统启动程序的实现

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ViewStartupApp()
```

```
{
    ListView3->Items->Clear();
    TListItem *mItem;
    TRegistry *reg;
    TStringList *tmplist=new TStringList();
    reg=new TRegistry();
    reg->RootKey=HKEY_LOCAL_MACHINE;
    reg->OpenKey("\\Software\\Microsoft\\Windows\\CurrentVersion\\Run\\",false);
    reg->GetValueNames(tmplist);
    for (int i=0;i<tmplist->Count;i++){
        mItem=ListView3->Items->Add();
        mItem->Caption=tmplist->Strings[i];
        mItem->SubItems->Add("Registry (Machine Run)");
        mItem->SubItems->Add(reg->
            ReadString(tmplist->Strings[i]));
    }
    reg->OpenKey("\\Software\\Microsoft\\Windows\\CurrentVersion\\RunServices\\",false);
    reg->GetValueNames(tmplist);
    for (int i=0;i<tmplist->Count;i++){
        mItem=ListView3->Items->Add();
        mItem->Caption=tmplist->Strings[i];
        mItem->SubItems->Add("Registry (Machine Service)");
        mItem->SubItems->Add(reg->
            ReadString(tmplist->Strings[i]));
    }
    delete tmplist;
    delete reg;
}
//---------------------------------------------------------------------------
```

删除启动程序的函数也是使用了 TRegistry 类进行操作，比较简单。在此不再列出具体代码。请读者自行参看配套光盘上本实例的源程序。

其实，在 Windows API 中也包含了操作注册表的一组函数，包括 RegCloseKey、RegConnectRegistry、RegCreateKey、RegDeleteKey、RegDeleteValue、RegEnumKey、RegEnumValue、RegFlushKey、RegGetKeySecurity、RegOpenKey、RegSaveKey、RegSetValue、RegUnLoadKey 等等，但实际上，使用 C++ Builder 提供的 TRegistry 类和 TRegIniFile 类来操作注册表更为方便和有效。

图 14-11 列出了注册表的启动程序。

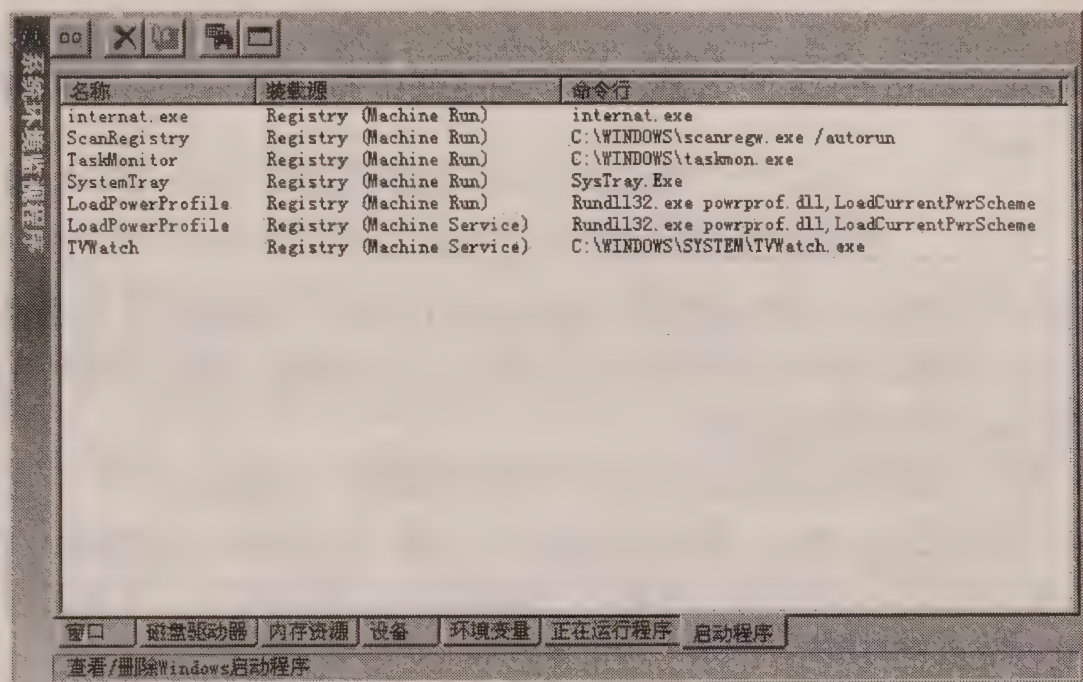


图 14-11 查看/删除系统启动程序

14.5.5 杀除进程

在系统环境监视程序中，提供随时关闭正在运行程序的功能，即杀除进程，在“窗口”和“正在运行程序”两个模块中都提供这项功能，也就是说，可以通过窗口和进程两种方式关闭正在运行的程序。

要杀除一个进程，必须获得该线程的父线程 ID（避免仅仅杀除子进程，这样可能会出现错误）。在“正在运行程序”模块中，我们已经获得父进程的 ID 即 pPid。所以在此模块中实现杀除进程是较为容易的。

Source 在“正在运行程序”模块实现杀除进程

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::KillProcess2()
{
    if (ListView2->SelCount==0){
        MessageBox(Handle,"请首先选择一个进程！",
            "中止进程",MB_OK|MB_ICONWARNING);
        return;
    }
    int pPid=StrToInt(ListView2->Selected->SubItems->Strings[0]);
    HANDLE ps = OpenProcess(1,false,pPid);
    if(ps&&TerminateProcess(ps,-9)){
        MessageBox(Handle,"成功中止进程！",
            "中止进程",MB_OK|MB_ICONINFORMATION);
```

```
    }
    else
        MessageBox(Handle,"中止进程失败！","中止进程",MB_OK|MB_ICONWARNING);
    ViewProcess();
}
//---------------------------------------------------------------------------
```

实现杀除进程功能的核心 API 函数是 TerminateProcess，但调用这个函数必须具备进程的句柄，所以，我们首先通过调用 OpenProcess 函数打开一个进程，获取该进程的句柄，而后再使用 TerminateProcess 函数杀除该进程。

在“窗口”模块中实现杀除进程功能基于同样的原理，只是多加了一个工作——通过窗口信息获得该窗口进程的 ID。为此，要首先获得窗口句柄，然后调用 GetWindowThreadProcessId 函数获取该窗口进程的 ID，此后的工作就与上述进行的工作相同了。下面是在“窗口”模块中实现杀除进程功能的关键代码：

```
HWND hWnd=FindWindow(
    ListView1->Selected->SubItems->Strings[1].c_str(),
    ListView1->Selected->Caption.c_str());
if (hWnd!=NULL){
    hg = GlobalAlloc(GMEM_SHARE,sizeof(DWORD));
    pPid = (DWORD *)GlobalLock(hg);
    result = GetWindowThreadProcessId(hWnd,pPid);
    if(result){
        HANDLE ps = OpenProcess(1,false,*pPid);
        if(ps&&TerminateProcess(ps,-9)){
            MessageBox(Handle,"成功中止进程！",
                "中止进程",MB_OK|MB_ICONINFORMATION);
            ViewWindow();
            GlobalUnlock(hg);
            GlobalFree(hg);
            return;
        }
        GlobalUnlock(hg);
        GlobalFree(hg);
    }
}
```

14.6 系统与设备

本节将实现范例程序中系统与设备监控部分的功能，这涉及到范例程序中的另外四个模块：“磁盘驱动器”、“内存资源”、“设备”和“环境变量”。

14.6.1 获取和设置驱动器信息

使用 Windows API 函数可以直接获取磁盘驱动器的有关信息。这一小节将讲解在 C++ Builder 中如何使用 Windows API 实现查看和设置磁盘驱动器的功能。

很显然，首先要做的工作是确定在用户的计算机上有多少个逻辑驱动器。API 函数 GetLogicalDrives 会告诉我们这一点，这个 API 调用无需参数，并且会返回一个 DWORD 类型值。这个值的每一个二进制位表示对应的驱动器是否存在，第 1 位（bit）对应驱动器 A，第二位对应驱动器 B 等等。同时，如果该位的值为“1”，则表示存在这个驱动器，“0”表示不存在。

所以，在本范例程序中，我们设计了如下算法来找出可用的逻辑驱动器。

Source 得出存在哪些逻辑驱动器

```
void __fastcall TMainForm::ViewDisk()
{
    DWORD drivers=GetLogicalDrives();
    for (int i=0;i<26;i++){
        if ((drivers&(1<<i))!=0){
            ComboBox1->Items->Add(AnsiString(char('A'+i))+":");
        }
    }
    ComboBox1->ItemIndex=1;
    ShowDiskInfo(); // 获取当前 ComboBox 所选驱动器的详细信息
}
```

这段程序是将 1 左移 i 位，得到一个只有在第 i 位是 1，其他位是 0 的数。将这个数和我们调用 GetLogicalDrives 得到的结果进行按位与计算，这样，如果 drivers 值的第 i 位是 1，则结果应该非 0，否则为 0。所以，通过判断按位与的结果是否为 0，就可以得到对应的驱动器是否可用。

接着调用 ShowDiskInfo 获取驱动器的详细信息。在这个函数中将完成这样几件事：第一，使用 C++ Builder 的 DiskSize 和 DiskFree 函数获得磁盘容量的详细信息（当然也可使用相应的 API 函数）；第二，利用 C++ Builder 的 Tchart 组件将磁盘容量信息以饼图形式表示出来；第三，调用 GetVolumeInformation 函数获取磁盘卷标、文件系统、磁盘序列号的信息；第四，调用 GetDriveType 函数得出当前驱动器的类型。这个函数的具体代码如下。

Source 获取逻辑驱动器的详细信息

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ShowDiskInfo()
```

```
{
    unsigned char num=ComboBox1->Items->
    Strings[ComboBox1->ItemIndex][1]-'A'+1;
    __int64 total=DiskSize(num);
    if (total==-1) total=0;
    __int64 free=DiskFree(num);
    if (free==-1) free=0;
    __int64 use=total-free;
    Label12->Caption=
        FormatFloat((AnsiString)"#,###,###,###' 字节'",total);
    Label9->Caption=
        FormatFloat((AnsiString)"#,###,###,###' 字节'",use);
    Label10->Caption=
        FormatFloat((AnsiString)"#,###,###,###' 字节'",free);
    Chart1->Series[0]->Clear();
    Chart1->Series[0]->Add(use,"已用空间",clBlue);
    Chart1->Series[0]->Add(free,"可用空间",clFuchsia);
    AnsiString path=ComboBox1->Items->
        Strings[ComboBox1->ItemIndex]+"\\";
    char volume[30],filesystem[30];
    DWORD seriel,mcl,flag;
    if (GetVolumeInformation(path.c_str(),volume,30,
        &seriel,&mcl,&flag,filesystem,30)){
    Edit1->Text=AnsiString(volume);
    Label8->Caption=IntToHex(int(seriel),8);
    Label6->Caption=AnsiString(filesystem);
    UINT drivetype= GetDriveType(path.c_str());
    switch (drivetype){
    case 1:
        Label4->Caption="不存在";
        break;
    case DRIVE_REMOVABLE:
        Label4->Caption="可移动磁盘";
        break;
    case DRIVE_FIXED:
        Label4->Caption="本机磁盘";
        break;
    case DRIVE_REMOTE:
        Label4->Caption="网络磁盘";
        break;
```

```
        case DRIVE_CDROM:
            Label4->Caption="光盘驱动器";
            break;
        case DRIVE_RAMDISK:
            Label4->Caption="内存虚拟磁盘";
            break;
        case 0:
            Label4->Caption="未知类型磁盘";
        }
        }
}
//---------------------------------------------------------------------------
```

图 14-12 给出了查看的磁盘驱动器的信息。

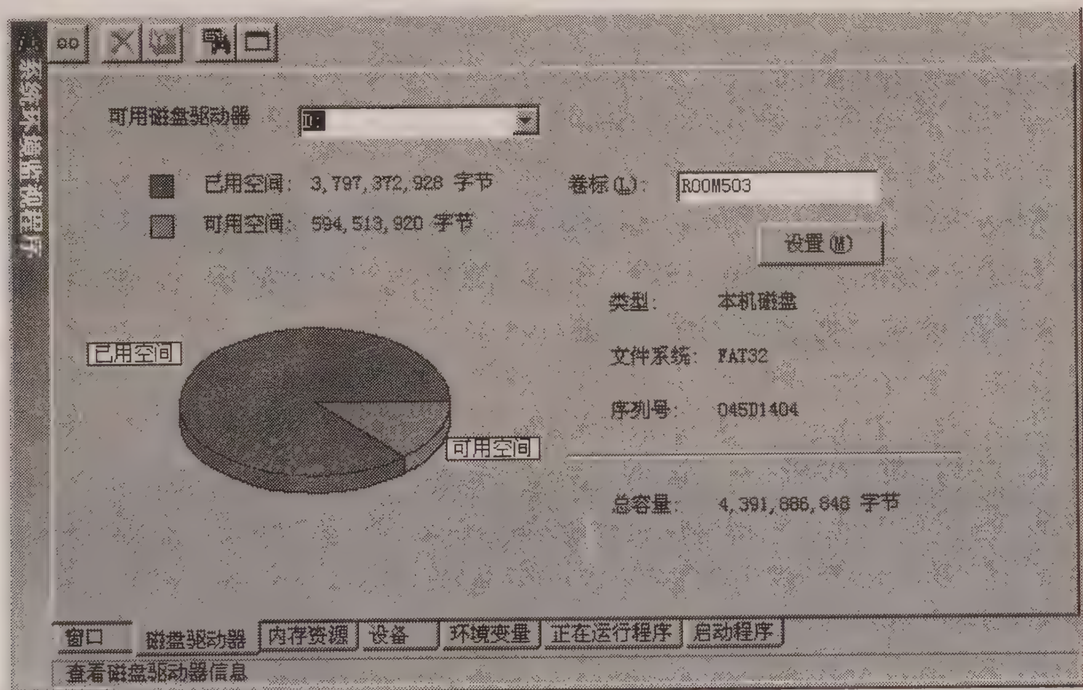


图 14-12　查看磁盘驱动器信息

在输出磁盘容量信息时，我们使用了 C++ Builder 的 FormatFloat 函数，这个函数的一个重要优点是可以定制数字的输出格式，类似于 TMaskEdit 组件的掩码。例如在本例中，程序对输出字节数加进了逗号分隔。

在获取磁盘容量部分中，我们使用了 64 位的整数数据类型__int64。读者可能觉得这个类型的关键字比较奇怪，在“int64”前加了“__”前缀。事实上，“__”前缀是为了区别标准 C++的关键字和 C++ Builder 中新增的关键字。如果觉得__int64 这样的变量声明不顺眼的话，也可以使用标准的 API 的 64 位整数声明 LONGLONG。

另外，程序中还使用了 C++ Builder 的 TChart 组件制作饼图。该组件的正式名称为 TeeChart，它是一个基于图表的组件，非常强大并且相当复杂。实际上，TeeChart 组件共有三个版本：Stand-alone 组件（在组件选项板的 Additional 页）、Data-aware 版（在 Data Controls 页）和 Report 版（在 QuickReport 页）。TeeChart 组件在数据库编程领域发挥着更大的作用。

TeeChart 组件仅仅为图表提供基本的结构和可视化容器，而真正的图表对象是 TChartSeries 类的子类。在本程序中，首先应该将 TChart 组件放置在窗体上，并且设置该组件，将不需要的 Legend（图例）、Title、Axis（轴）去掉。然后建立一个新的饼图的 Series。这样在程序中就可以使用 TChartSeries 类的 Add 方法为图表添加新的数据。

在完成磁盘容量信息的显示之后，继续调用 API 函数 GetVolumeInformation 来获取磁盘驱动器的其他信息。这个函数要求提供该磁盘的根目录，并可返回卷标、序列号、最大文件名长度（该磁盘是否支持长文件名）、文件系统标志（如是否是使用 DoubleSpace 的压缩盘）、文件系统（如 FAT32、NTFS）等信息。在程序中我们仅显示了磁盘的卷标、序列号和文件系统。

最后，程序调用另一个与驱动器相关的 API 函数 GetDriveType，用于获取当前驱动器的类型。

磁盘的卷标是可以设置的。这个模块提供设置磁盘卷标的功能，这也是通过调用 APISetVolumeLabel 来实现的。

14.6.2 获取内存资源信息

通过简单的调用 API 函数，就可以知道系统当前内存资源的详细情况。实现这一部分功能的关键 API 是 GlobalMemoryStatus 函数。在本范例中，同样使用 TeeChart 图表组件将内存信息显示出来。下面是这部分功能的实现代码。

Source 获取内存资源信息

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ViewMem()
{
        MEMORYSTATUS MS;
        MS.dwLength= sizeof(MS);
        GlobalMemoryStatus(&MS);
        Label21->Caption=
                FormatFloat((AnsiString)"#,###' KB'",MS.dwTotalPhys/1024);
        Label23->Caption =
                IntToStr(MS.dwMemoryLoad)+" %";
        int use,free,total;
        total=int(MS.dwTotalPhys/1024);
        free=int(MS.dwAvailPhys/1024);
        use=total-free;
        Chart2->Series[0]->Clear();
        Chart2->Series[0]->Add(free,"",clAqua);
        Chart2->Series[0]->Add(use,"",clBlue);
```

```
    Label26->Caption=
        FormatFloat((AnsiString)"#,###' KB'",total);
    Label27->Caption=
        FormatFloat((AnsiString)"#,###' KB'",free);
    Label32->Caption=
        IntToStr(int(float(free)/total*100))+"%";
    total=int(MS.dwTotalVirtual/1024);
    free=int(MS.dwAvailVirtual/1024);
    use=total-free;
    Chart3->Series[0]->Clear();
    Chart3->Series[0]->Add(free,"",RGB(255,128,0));
    Chart3->Series[0]->Add(use,"",clRed);
    Label30->Caption=
        FormatFloat((AnsiString)"#,###,###' KB'",total);
    Label31->Caption=
        FormatFloat((AnsiString)"#,###,###' KB'",free);
    Label33->Caption=
        IntToStr(int(float(free)/total*100))+"%";
}
//---------------------------------------------------------------------------
```

图 14-13 给出了内存资源信息。

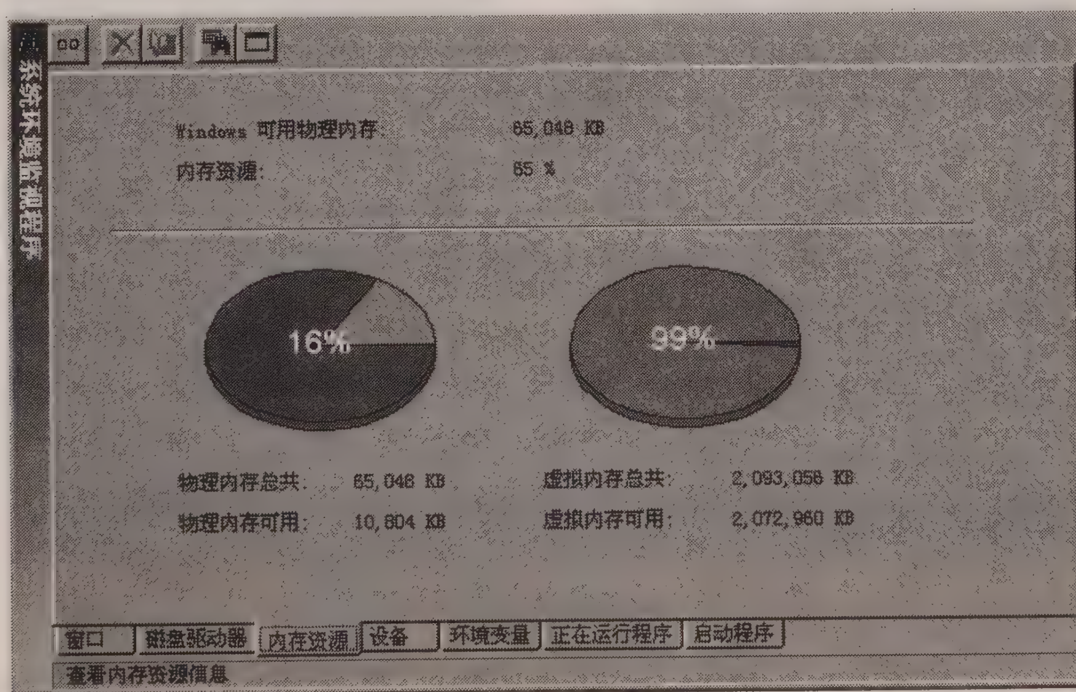


图 14-13　查看内存资源信息

GlobalMemoryStatus 函数需要一个 MEMORYSTATUS 的结构作为引用参数。MEMORYSTATUS 结构声明如下：

```
typedef struct _MEMORYSTATUS { // mst
    DWORD dwLength;          // 该结构的大小，sizeof(MEMORYSTATUS)
    DWORD dwMemoryLoad;  // 当前使用内存资源的百分比
```

```
    DWORD dwTotalPhys;      // 物理内存
    DWORD dwAvailPhys;      // 可用物理内存
    DWORD dwTotalPageFile;  // 文件页面
    DWORD dwAvailPageFile;  // 可用文件页面
    DWORD dwTotalVirtual;   // 虚拟内存
    DWORD dwAvailVirtual; // 可用虚拟内存
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

本范例中，通过绘制饼图显示了物理内存和虚拟内存信息。

14.6.3 获取设备信息与动态调整显示

在这一模块中，将演示如何调用 API 获得计算机和显示器设备信息，同时，还将演示如何使用 API 动态调整显示分辨率和颜色数。

获得计算机信息的 API 调用较为简单，涉及到 GetComputerName 和 GetSystemInfo 两个函数。GetComputerName 用于获得计算机在网络中的名字，相应的有函数 SetComputerName，用于设置计算机名。通过 GetSystemInfo 函数可以获得 CPU 信息，该函数需要一个 SYSTEM_INFO 的结构作为引用参数。该结构的 wProcessorLevel 域可返回 CPU 的级别，图 14-14 给出了设备信息。

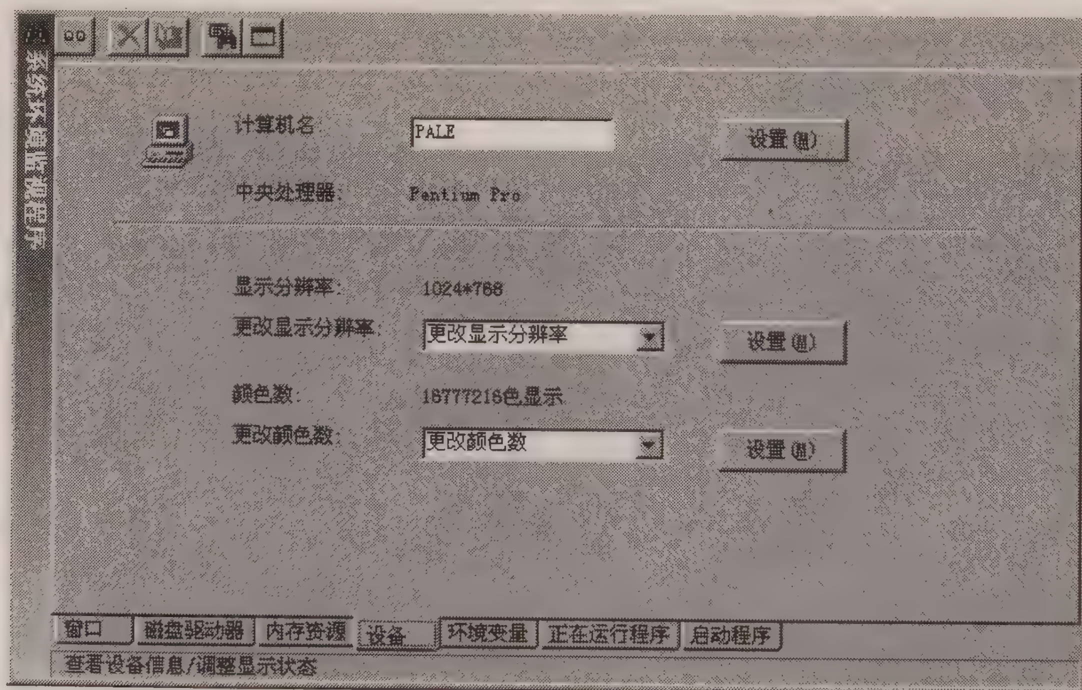


图 14-14 设备信息查看和设置模块

获得显示信息可以使用 EnumDisplaySettings 函数，这个函数是这样声明的：

```
BOOL EnumDisplaySettings(
  LPCTSTR lpszDeviceName,  // 指定显示器
  DWORD iModeNum,          // 指定要获取的图像模式
  LPDEVMODE lpDevMode      // 指向返回信息结构的指针
  );
```

lpszDeviceName 指定显示器，当只有一个显示器时，该参数为 NULL。iModeNum 指定

一个获取信息的索引值，如果要获取全部显示信息，应该在一个 while 循环中让 IModeNum 从 0 开始递增并不断调用 EnumDisplaySettings，直至该函数返回 false 为止。第三个参数指向一个 DEVMODE 结构，这个结构非常庞大和复杂，其中以下几个域是较为常用的：

dmBitsPerPel	像素颜色位数。2 的 dmBitsPerPel 次方是实际显示颜色数。
dmPelsWidth	返回显示器以像素为单位的宽度。
dmPelsHeight	返回显示器以像素为单位的高度。
dmDisplayFrequency	返回显示刷新频率。

下面是实现获取计算机和显示器信息的代码：

Source 获取计算机和显示器信息

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::ViewDevice()
{
        char Computer[MAX_COMPUTERNAME_LENGTH+1];
        DWORD size=MAX_COMPUTERNAME_LENGTH+1;
        GetComputerName(Computer,&size);
        Edit3->Text=AnsiString(Computer);
        SYSTEM_INFO SI;
        GetSystemInfo(&SI);
        switch (SI.wProcessorLevel){
        case 3:
            Label19->Caption="Intel 80386";
            break;
        case 4:
            Label19->Caption="Intel 80486";
            break;
        case 5:
            Label19->Caption="Pentium";
            break;
        case 6:
            Label19->Caption="Pentium Pro";
            break;
        default:
            Label19->Caption="未知类型";
        }
        DEVMODE DevMode;
        int i=0;
        while(EnumDisplaySettings(NULL,0,&DevMode))
         i++;
```

```
            Label34->Caption=AnsiString(DevMode.dmPelsWidth)+
    "*"+AnsiString(DevMode.dmPelsHeight);
            DWORD Colors = 1 << DevMode.dmBitsPerPel;
            Label39->Caption = AnsiString(Colors)+"色显示";
    }
    //---------------------------------------------------------------------------
```

动态调整显示需要调用 ChangeDisplaySettings 函数，该函数同样需要一个 DEVMODE 结构作为参数，程序中应该填写 DEVMODE 结构的 dmFields 域，告诉 ChangeDisplaySettings 函数需要改变显示器的哪方面属性。例如，下面的代码是在用户选择了 ComboBox 中的一项后，改变色彩信息的函数：

Source 动态调整显示颜色数

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::Button5Click(TObject *Sender)
{
        DEVMODE DevMode;
        DevMode.dmFields=DM_BITSPERPEL;
        int bitspixel;
        switch (ComboBox3->ItemIndex){
        case 0:
            bitspixel=4;
            break;
        case 1:
            bitspixel=8;
            break;
        case 2:
            bitspixel=16;
            break;
        case 3:
            bitspixel=24;
        }
        DevMode.dmBitsPerPel=bitspixel;
        if(ChangeDisplaySettings(&DevMode,0)
            ==DISP_CHANGE_SUCCESSFUL)
        MessageBox(Handle,("成功设置显示色彩数为"+
                IntToStr(1<<bitspixel)+"色。").c_str(),
                "设置显示模式",MB_OK|MB_ICONINFORMATION);
}
//---------------------------------------------------------------------------
```

调整显示分辨率与调整颜色数类似，但首先应该设定 dmFields 为 DM_PELSWIDTH |DM_PELSHEIGHT，并设置 dmPelsWidth 和 dmPelsHeight 对应需要改变的显示分辨率。

14.6.4 获取和设置系统环境变量

环境变量包含了一系列与 Windows 系统相关的参数。虽然，在 Win32 中环境变量的许多内容已经被注册表所取代，但是 Path 等重要系统参数仍使用环境变量管理。在实际编程中，可能会遇到需要读写环境变量的时候，有两个 Windows API 函数和系统环境变量操作相关：GetEnvironmentStrings 和 SetEnvironmentVariable 函数，分别对应环境变量的读写操作。

以下是在本范例程序中，将所有环境变量信息显示在一个 TListBox 组件的代码。

Source 显示所有环境变量

```
void __fastcall TMainForm::ViewEString()
{
        ListBox1->Items->Clear();
        char *strings=GetEnvironmentStrings();
        for (int index=0;strings[index]!=0;index+=strlen(&strings[index])+1)
            ListBox1->Items->Add(&strings[index]);
}
```

GetEnvironmentStrings 函数返回指向当前环境变量内存块的指针。在环境变量内存块中，每一个环境变量字符串依次排列，所以程序中使用如下的 for 循环来遍历每一个环境变量字符串：

```
for (int index=0;strings[index]!=0;index+=strlen(&strings[index])+1)
```

每一个环境变量字符串以“环境变量名=值”格式存储，如图 14-15 所示。

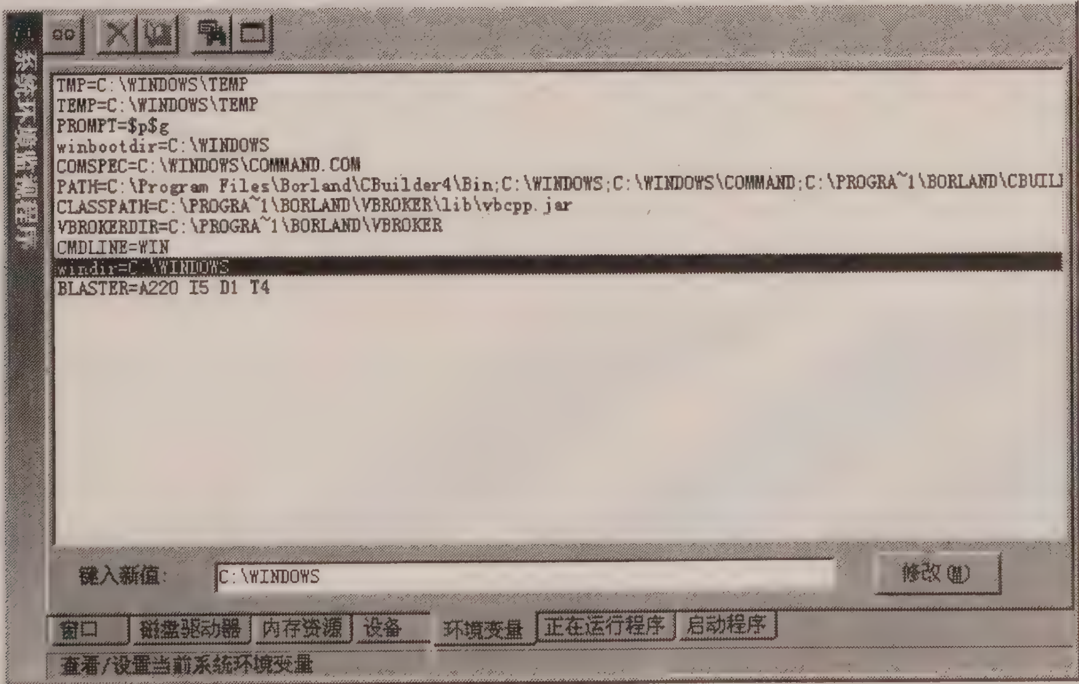


图 14-15 查看/设置系统环境变量

改变环境变量的值使用 SetEnvironmentVariable 函数，以下是实现修改环境变量的代码。

```
void __fastcall TMainForm::Button2Click(TObject *Sender)
{
        SetEnvironmentVariable(
            ListBox1->Items->Names[ListBox1->ItemIndex].c_str(),
            Edit2->Text.c_str());
}
```

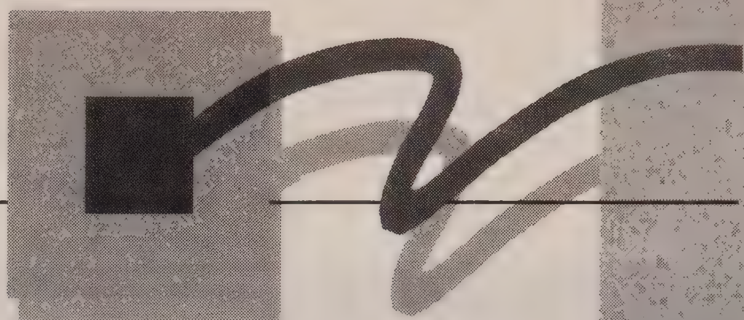
使用 GetEnvironmentStrings 可以获得全部环境变量的值，但实际上我们往往并不需要知道全部环境变量值，而仅仅需要获取特定的环境变量值，这时则可以使用 GetEnvironmentVariable 函数。下面的代码演示了 GetEnvironmentVariable 函数的调用，等价于使用 GetTempPath 函数。

```
char Buf[100];
GetEnvironmentVariable("TEMP",Buf,100);
```

第 15 章

BCB 抓图大师
——高级 DLL 技术和钩子函数

DLL 彻底研究
插件技术及其实现
通过内存映像文件共享数据
使用 Windows 钩子函数
截图程序实例创建



本章导读

本章主要实例是名为“BCB 抓图大师”的实用截图工具。“BCB 抓图大师”能够实现多种截图方式和多图像格式的屏幕截图，同时具备截取图像的浏览功能。通过该程序的实现，本章系统讲述了 Windows 钩子函数的使用以及进程间利用映像文件共享数据的技术。

另外，本章还将讲述近年来一项热门技术——插件（Plug-In）技术的编程实现，并完成一个实现功能模块“即插即用”的应用程序。

本章主要内容包括：

- 以动态链接方式调用 DLL、动态链接库的生存周期。
- 插件技术及其实现。创建一个完整的应用插件的实例程序——图像特效处理程序。
- 进程间的数据共享，内存映像文件（Memory Mapped File）。
- 利用 Windows 钩子函数技术创建实例程序“BCB 抓图大师”，并演示使用内存映像文件共享数据的技术。

15.1 DLL 彻底研究

Windows 的执行文件可以划分为两种形式：程序和动态链接库（DLL）。DLL 在 Windows 编程中具有举足轻重的地位。

本书前一章已经示范了基础 DLL 的编写，并演示了如何使用静态加载方式调用 DLL。然而那一部分内容对于 DLL 来说是较浅显的。下面将针对 DLL 做彻底的探讨，从而使您对动态链接库技术有一个全面的认识。

15.1.1 动态加载 DLL

第 14 章中以静态加载方式实现了 DLL 调用，但有时应用程序自己加载 DLL 可能更好些。如果开发者希望指定用户可以使用哪一个 DLL，或通过用户的选择使用不同的 DLL，这时可以使用动态加载 DLL 实现函数调用。动态加载过程允许使用不同的 DLL 来达到不同的目的，动态加载方式为编程带来了极大的灵活性。

为了动态加载 DLL，必须把它映像到相应的应用进程地址空间，可通过调用 Win32 API 函数 LoadLibrary 或 LoadLibraryEx 来完成映像。LoadLibrary 函数需要一个参数，即模块文件名。可以这样使用：

```
HINSTANCE hInst = LoadLibrary("Mydll");
if (hInst==NULL){
        ……// DLL 模块加载失败，错误处理
```

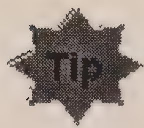
如果不加指定，Windows 系统假定缺省文件扩展名为“.dll”，在上面的代码中，系统将寻找“Mydll.DLL”文件。Windows 系统将从进程所在目录开始，并沿 PATH 环境变量指定的路径寻找该文件，除非指定路径。

一旦找到文件，Windows 将用已在内存的 DLL 路径全名和找到文件的路径全名比较，如果匹配（确定为同一 DLL），则返回库句柄，而不是加载库的拷贝。

LoadLibrayEx 函数是 LoadLibrary 函数的加强版本，它和 LoadLibrary 函数基本相同。LoadLibraryEx 函数的声明是这样的：

```
HINSTANCE LoadLibraryEx (
  LPCTSTR lpLibFileName,        // 加载模块文件名
  HANDLE hFile,                 // 保留参数，必须为 NULL
  DWORD dwFlags            // 进入点执行标记
);
```

LoadLibraryEx 函数同样在 lpLibFileName 参数得到 DLL 文件的名字。hFile 为保留参数，目前无用，dwFlags 参数为 DWORD 值，该值说明了三个可用标志位的一个。



除了动态链接库之外，LoadLibrary 和 LoadLibraryEx 函数也支持加载可执行文件到内存。例如可以使用 LoadLibrary 加载 Win32 应用程序文件，使用 LoadResouce 函数提取图标等资源。LoadLibrary 函数返回句柄在调用 FindResouce 和 LoadResouce 时用到。

如果 DLL 成功的被加载，接着就可以着手进行下一步工作——找到 DLL 中需要使用函数的地址。Win32 函数 GetProcAddress 可以做这件事情。它需要一个由 LoadLibray 函数返回的句柄和函数名。这个名字应该是.DLL 文件中的输出函数名。

如果没有找到函数，GetProcAddress 将返回 NULL；如果找到指定函数，它返回函数的指针。应用程序还应该保证在使用指针时使用与 DLL 中函数相同的返回值和参数列表。

下面的一段代码动态加载方式导入了上一章创建的 SelDir.dll 动态链接库，并找到 SelectDirectory 函数，然后调用该函数。

```
bool (_stdcall *SelectDir)(LPCTSTR,LPCTSTR,LPTSTR)=NULL;
HINSTANCE hInst = LoadLibrary("SelDir.dll");
if (hInst)
{
        (FARPROC)SelectDir=GetProcAddress(hInst,"SelectDirectory");
        if (SelectDir)
        {
                char Dir[100];
                if (SelectDir("DLLDemo","请您选择一个目录",Dir))
                        MessageBox(0,("您选择了"+AnsiString(Dir)).c_str(),
                                "DLLDemo2",MB_OK);
        }
}
```

这段代码和前一章中完成同样功能的代码不同，编译器和链接器会忽视 DLL 存在与否。这样应用程序和 DLL 之间完全分开了（不需要在工程中加入相应.lib 文件）。当运行程序时，会马上装载 DLL。甚至可以更为灵活，让用户输入所需的 DLL 名称。在一些情况下，这样做有很大的好处。

程序中首先声明函数指针类型。显然，我们必须和 DLL 中的声明保持一致。假如，向 SelectDirectory 函数传递了三个 int 值，当通过指针调用函数时，有可能破坏堆栈，这对应用程序是毁灭性的。

动态加载 DLL 的方式允许应用程序在运行时关闭 DLL，这是通过调用 FreeLibrary 函数完成的，这个函数需要一个模块实例句柄作为参数。

图 15-1 是一个动态导入链接库的示例，动态加载了 SelDir.dll 并调用其中的 SelectDirectory 函数。

这里需要说明的是，无论如何 DLL 都是动态链接的，不管这个 DLL 时动态加载还是静态加载。这里的区别仅仅是隐式的装载 DLL 或显式的装载 DLL。应该说，隐式装载 DLL

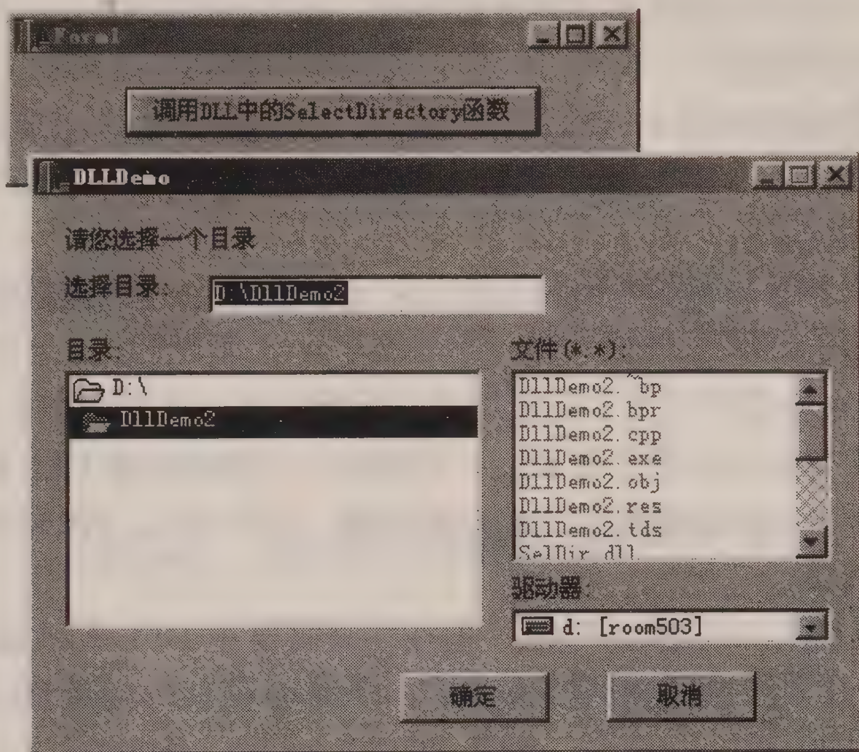


图 15-1　动态导入动态链接库示例

（静态）的方式更可靠，更安全，速度也快一些。但显式装载 DLL 的方式真正体现了 DLL 的灵活性，它的功能确实很强大。

15.1.2　DLL 入口点及生存周期

1. DLL 入口点函数

在 C++ Builder 中，DLL 程序的入口点是称之为 DLLEntryPoint 的函数。概括来讲，它指定了 DLL 执行时启动代码查找函数。如果 DLLEntryPoint 存在，则在每次 DLL 引用（附加一个进程或线程）时调用此函数，同时，每当一个线程或进程离开时也调用该函数。DLLEntryPoint 函数担当 DLL 的入口和出口例程的角色，通常用来完成一些初始化操作和清理任务。使用 C++ Builder 的 DLL 向导创建的 DLL 框架中已经包含了这个函数：

```
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    return 1;
}
```

DLLEntryPoint 函数传递三个参数，其中第三个参数是保留参数，目前无用。hinst 参数为 DLL 模块实例的句柄。其值是 DLL 模块的基地址。DLL 模块的 HINSTANCE 等同于 DLL 模块的 HMODULE。

reason 参数是 DLLEntryPoint 函数的非常重要的参数。它指出 DLL 被调用时传递的标志。此参数为下面值中的一个：

DLL_PROCESS_ATTACH

当某个进程初次将 DLL 映像到它的地址空间时（该 DLL 第一次被引用），将传递这个

值。DLL 可以利用这个机会初始化全局对象或变量，并完成首次 DLL 引用的特定处理。

DLL_THREAD_ATTACH

当前进程创建新的线程时传递该标志。DLL 可以利用这个机会在后继线程附加时完成特定的处理。

DLL_THREAD_DETACH

在装载进程中的一个线程已结束时，传递该标志。此时 DLL 会释放分配单元，并把相关的标记位和计数器复位。

DLL_PROCESS_DETACH

当前系统中的所有进程都卸载了该 DLL，在最后一个进程卸载 DLL 时，DLLEntryPoint 传递这个标志。可以利用这个机会释放相关的全局变量，关闭 DLL_PROCESS_ATTACH 期间打开或初始化的文件或端口。

尽管 DLL 会多次调用 DLLEntryPoint，但往往只需要在 DLL 初次被装载和 DLL 最终撤离应用时进行处理工作，此时只需根据系统传入的 reason 值就可判断 DLL 当前的调用状态。

2. 引用计数和生存周期

前面提到 DLL 载入及释放并未给出明确的解释，DLL 的载入和卸载牵涉到多个进程使用时的加载和卸载。由于 DLL 是动态链接的，因此常常同时存在许多应用程序在使用同一个 DLL（如 Windows 的 User32.dll，从 Windows 启动开始，无时无刻应用程序不在调用它），举例来说：如果一个 MyDll.DLL 同时被 A、B、C 三个应用程序使用，则 MyDll.DLL 会被载入三次。然而系统为了有效利用资源，并不会重复载入这个 DLL，因此此时在系统内会有一个表格来记载 MyDll.DLL 的使用次数。所以当 A 应用载入 MyDll.DLL 后，B、C 应用再次载入 MyDll.DLL 时，此时系统只是将 MyDll.DLL 的引用次数加一，然后将先前载入的 MyDll.DLL 位址映像到 B、C 进程，如此就可以达到共享 DLL 的目的了。同样地在释放 MyDll.DLL 时，若该 DLL 同时有多个进程在使用，系统纯粹只是将该 DLL 的使用次数减一，当其使用次数等于 0 时，系统才会“真正”地将它由系统中释放。



实际的 DLL 装载和卸载过程比上面讨论的要复杂。诸如主线程的创建，每个 DLL 入口点的调用，函数引用的析址，装载到预申请的基地址，重定位等等。但 DLL 运行的大体机制是较为简单的。

由上可知， DLL 的载入及释放都和它的引用计数有关。所以 DLL 的生存周期也就在 DLLEntryPoint 被 DLL_PROCESS_ATTACH 调用和 DLL_PROCESS_DETACH 调用之间。引用计数器为 0 时，系统才会以 DLL_PROCESS_DETACH 调用 DLLEntryPoint，这种 reason 参数的通知时系统根据 DLL 的引用计数器自动实现的。

3. DLL 内存

与静态库不同，DLL 不能够像静态库那样有效地变成程序代码的一部分。在 Windows 32 系统中，动态链接库处理内存的方法和 16 位 Windows 并不相同，在 Win 16 系统中，DLL 的内存空间不属于进程空间，这样各个进程之间可以共享 DLL 内存，这就意味着可能通过 DLL

简单的交换数据。

在 Win32 系统中，DLL 内存被映射到进程空间中，每个进程都有自己的全局内存拷贝；加在 DLL 的每一个新的进程都重新初始化这一内存区域，这意味着如果进程 A 装载了一个 DLL，同时进程 B 也装载了同一 DLL，但映射到进程 A 的 DLL 的全局和堆分配数据对于进程 B 来讲是不可存取和不可使用的。

虽然 DLL 的代码只装载一次，但每份 DLL 在调用应用程序的地址空间都有自己的数据。不过 Win32 还是提供了实现全局数据共享的技术，用于突破进程的界限。用于数据共享技术中，最常使用的是内存映像文件。在本章稍后会详细讨论这个问题。

15.2 插件（Plug-In）技术

插件技术在目前的 Windows 应用程序开发中是一项非常热门的技术，在很多著名软件中被使用，如 Photoshop、Winamp、Adobe Acrobat 等等。实际上，所谓插件技术并不是什么新的东西，它们本身也并不复杂。插件技术认为是一种新的编程思想而非新技术可能更合适一些。

15.2.1 插件技术分析

事实上，插件技术是 DLL 动态链接库的直接应用。插件（Plug-In），其实就是一个个 DLL 模块文件，不管它是如何命名的。例如在 Photoshop 5 中插件是以.8bf 等扩展名命名，但其实这些插件还是标准的 DLL，可以试着改变这些文件的扩展名，然后用“快速查看”工具了解这些 DLL 的结构。

插件技术实现了应用程序功能模块的“即插即用”。一般来说，如果用户拷贝一个插件到支持插件的应用程序的插件目录中，这个应用程序就能够自动识别并启用这个插件。插件完成某个特定的功能模块，并和应用程序协同工作。然而插件文件和应用程序是分离的，并且插件是一个可执行程序模块，显然只有插件实现为一个 DLL 才能和应用程序结合起来。

插件的载入是预先不可知的。应用程序必须在运行时查找可用的插件（DLL），然后装载插件，并调用插件模块中的函数。由于这种不可知性，我们只能够以动态加载方式装入插件，动态链接库（DLL）恰好可以实现这样的调用方式，即使用 LoadLibrary 和 FreeLibrary 函数。

另外，为了使应用程序能够正确的调用插件模块功能，必须规定一个应用程序和插件 DLL 之间的通信方式。一般来说，插件必须实现一个规定名称的身份确认函数，如 GetMyPlugInName。当应用程序找到一个 DLL 并装载之后，应用程序会在 DLL 中查找名为 GetMyPlugInName 的函数并调用这个身份确认函数。如果一切正常，该插件将被认为是有效插件模块，此时应用程序会做一系列工作将该插件“嵌入”程序，例如在菜单中添加调用插件功能的菜单项等等，并为调用插件功能做准备。

插件和应用程序的规则还包括功能函数的统一命名和统一参数类型，例如 Photoshop 5 中

所有.8bf 的插件的导出函数都为 AboutDialogProc、DrawBMPProc、PreviewWndProc、DoParameters 等几个。当然，也可以和插件模块通信获取特定函数名，总之必须实现程序和插件的共同遵循的规则。

15.2.2 插件程序实例

插件技术提供功能模块“即插即用”的优秀特性，这样就可以方便的为程序增加新功能、新特性，为应用程序带来无限的可扩展性。这一切都得益于 DLL 动态链接库的强大功能。

本节将创建一个简单的使用插件技术的应用程序，并通过该范例的创建讲解实现插件技术应用程序编程的具体方法。这个实例是一个图像特效处理程序，所有的特效处理功能都是基于 DLL 插件实现的。

1. 插件模块

在实例程序中，所制定的插件规范是，插件必须具备两个函数。第一个函数为 GetPlugInEffectName，这是一个用于身份认证的函数，同时该函数必须有两个 LPTSTR 字符串指针参数，用于该函数返回插件的名称和相关信息；第二个函数是插件的真正执行模块，函数名规定为 DoEffect，此函数具有 TBitmap 指针类型的参数。在 DoEffect 函数中实现具体的图像特效处理，例如对图像进行灰阶转换、亮度调整、对比调整、模糊滤镜等等。下面的例子是反色特效插件的实现代码。

Source 插件“反色效果”的实现代码

```
extern "C" __declspec(dllexport)
void __stdcall GetPlugInEffectName(LPTSTR Effect,LPTSTR Info);
extern "C" __declspec(dllexport)
void __stdcall DoEffect(Graphics::TBitmap *Bmp);
//---------------------------------------------------------------------------
void __stdcall GetPlugInEffectName(LPTSTR Effect,LPTSTR Info)
{
        strcpy(Effect,"反色效果");
        strcpy(Info,"Palesoft Inverse Color Plug-In v1.0   [INVERSE.DLL]");
}
//---------------------------------------------------------------------------
void __stdcall DoEffect(Graphics::TBitmap *Bmp)
{
        BYTE *ptr;
        Bmp->PixelFormat=pf24bit;
        for (int y = 0; y < Bmp->Height; y++)
```

```
        {
            ptr =(BYTE *) Bmp->ScanLine[y];
            for (int x = 0; x < Bmp->Width*3; x+=3){
                ptr[x]=BYTE(255-ptr[x]);
                ptr[x+1]=BYTE(255-ptr[x+1]);
                ptr[x+2]=BYTE(255-ptr[x+2]);
            }
        }
}
//---------------------------------------------------------------------------
```

这个插件的代码较为简单。实际上，还可以制作更为复杂的特效，并在插件中加入特效执行的控制对话框，在 DoEffect 函数中使用对话框。在本书光盘中还有一个灰阶转换特效的插件程序，使用的数字图像处理技术在本书的第 5 章中已经讲述。

2. 使用插件技术的应用程序

在主程序中，为了实现插件和程序的接口，还需要作一系列具体工作，在“图像特效处理”范例程序中，完成下面的步骤：

(1) 查找规定的插件所在目录（程序所在目录的 Plug-Ins 子目录）的所有文件，对于找到的.DLL 文件进行装载。
(2) 在找到的.DLL 文件中查找 GetPlugInEffectName 函数。如果找到该函数，则调用函数获得插件实现的效果名称和插件信息。如果上述操作失败，则卸载这个 DLL。
(3) 将效果名称添加到菜单项，并在插件设置对话框中添加插件信息，为正确调用插件功能做好准备。
(4) 重复上述过程直至 Plug-Ins 目录查找全部完成。

图 15-2 是图像特效处理范例程序，该程序找到并装载了两个插件。

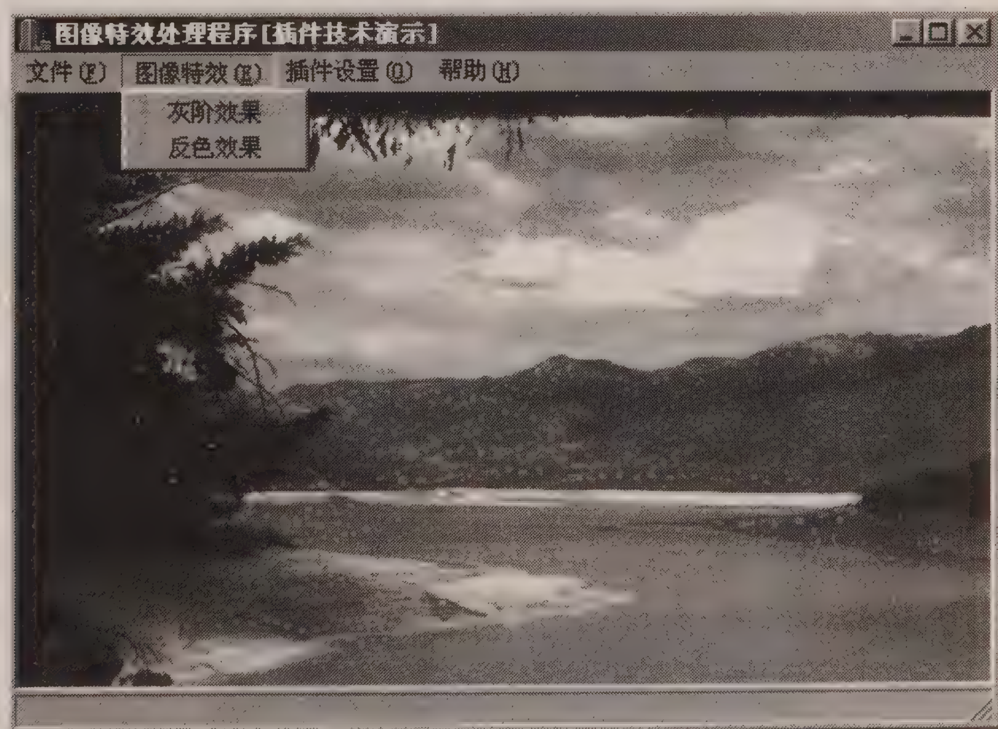


图 15-2　图像特效处理范例程序

具体实现代码如下：

Source 在程序中寻找并装载插件

```
void __fastcall TMainForm::LoadPlugIns()
{
        void (__stdcall *GetName)(LPTSTR,LPTSTR)= NULL;
        char Effect[100],Info[100];
        TSearchRec sr;
        int i=0;
        AnsiString Path=ExtractFilePath(Application->ExeName)+"Plug-Ins\\*.*";
        if (FindFirst(Path,0, sr) == 0){
        if (ExtractFileExt(sr.Name).UpperCase()==".DLL"){
                hInsts[i] = LoadLibrary( ("Plug-Ins\\"+sr.Name).c_str() );
                (FARPROC)GetName=
                        GetProcAddress(hInsts[i],"GetPlugInEffectName");
                if (GetName){
                        i++;
                        GetName(Effect,Info);
                        TMenuItem *NewItem = new TMenuItem(E1);
                        NewItem->OnClick=EffectsClick;
                        NewItem->Caption = AnsiString(Effect);
                        E1->Add(NewItem);
                }
                else FreeLibrary(hInsts[i]);
        }
        while (FindNext(sr) == 0)
        {
        if (ExtractFileExt(sr.Name).UpperCase()==".DLL"){
                hInsts[i] = LoadLibrary( ("Plug-Ins\\"+sr.Name).c_str() );
                (FARPROC)GetName=
                        GetProcAddress(hInsts[i],"GetPlugInEffectName");
                if (GetName){
                        i++;
                        GetName(Effect,Info);
                        TMenuItem *NewItem = new TMenuItem(E1);
                        NewItem->OnClick=EffectsClick;
                        NewItem->Caption = AnsiString(Effect);
                        E1->Add(NewItem);
                }
```

```
                else FreeLibrary(hInsts[i]);
            }
        }
    }
    Count=i;
    FindClose(sr);
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::EffectsClick(TObject *Sender)
{
    int Pos;
    void (__stdcall *DoEffect)(Graphics::TBitmap *Bmp)= NULL;
    TMenuItem *ParentItem;
    ParentItem=(TMenuItem *)((TMenuItem *)Sender)->Parent;
    Pos=ParentItem->IndexOf((TMenuItem *)Sender);
    hInsts[Pos];
    (FARPROC)DoEffect=GetProcAddress(hInsts[Pos],"DoEffect");
    if (DoEffect){
        Graphics::TBitmap *tmpBmp=new Graphics::TBitmap();
        tmpBmp->Assign(Image1->Picture->Graphic);
        DoEffect(tmpBmp);
        Image1->Picture->Graphic=tmpBmp;
        delete tmpBmp;
    }
}
//---------------------------------------------------------------------------
```

LoadPlugIns 函数实现插件的查找和装入，程序中使用 FindFirst/FindNext/FindClose 块在 Plug-Ins 目录查找插件。找到插件后进行相关的处理工作，其中 hInsts 数组用于纪录可用插件模块句柄。以便于调用插件中的效果处理函数。对于找到的插件，通过操纵菜单加入相应的下拉菜单项，并为菜单项连接 OnClick 执行函数。图 15-3 为插件设置对话框，显示了当前装载的插件信息。

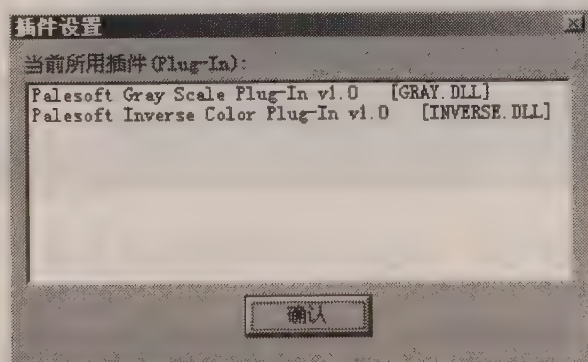


图 15-3 插件设置对话框

EffectsClick 函数为插件效果菜单项的通用单击响应函数。在函数中判断了所选菜单项在下拉菜单中的位置，并根据位置在不同的插件中调用 DoEffect 函数进行特效处理。

15.3 实例创建分析

本章将实现一个功能丰富的截图工具——“BCB 抓图大师”。对于一个截图工具来说，必须完成的一项工作是实现截图热键功能。在这个实例程序中，将使用 Windows 钩子（Hook）函数，为 Windows 系统安装全局键盘消息钩子来实现热键操作。当然，这并不是唯一的热键实现方式，这样做是为了演示 Windows 钩子函数这一强大工具的使用。

钩子函数的确非常强大，它类似 DOS 编程中的中断捕获。钩子函数高于 Windows 进程，它被嵌入在 Windows 的消息系统。钩子函数可以控制各种各样的 Windows 消息：捕获消息、对消息进行处理甚至吞掉消息。

对于系统钩子函数来说，必须以 DLL 的形式来实现。但与一般的 DLL 调用不同，对于钩子函数来说，存放钩子回调函数的 DLL 是安装到系统中，并由系统调用，而不是像一般的 DLL 调用那样由应用程序实现。

另一个问题是，实际编程中必须将在应用程序中进行的功能设置信息传给钩子函数，因为存放钩子回调函数的 DLL 是由系统调用的，所以直接将参数传递给 DLL 全局变量是不可行的。映射到应用程序的一份 DLL 和由系统调用那份 DLL 的全局变量是不同的，每份 DLL 在调用应用程序的地址空间中都有自己的数据。所以必须以特殊的通信方式向系统钩子传递设置信息。本范例程序使用了内存映像文件技术，使用内存映像文件技术可以创建真正的系统范围的全局变量。

图 15-4 是处于设计期间的范例程序主窗体。

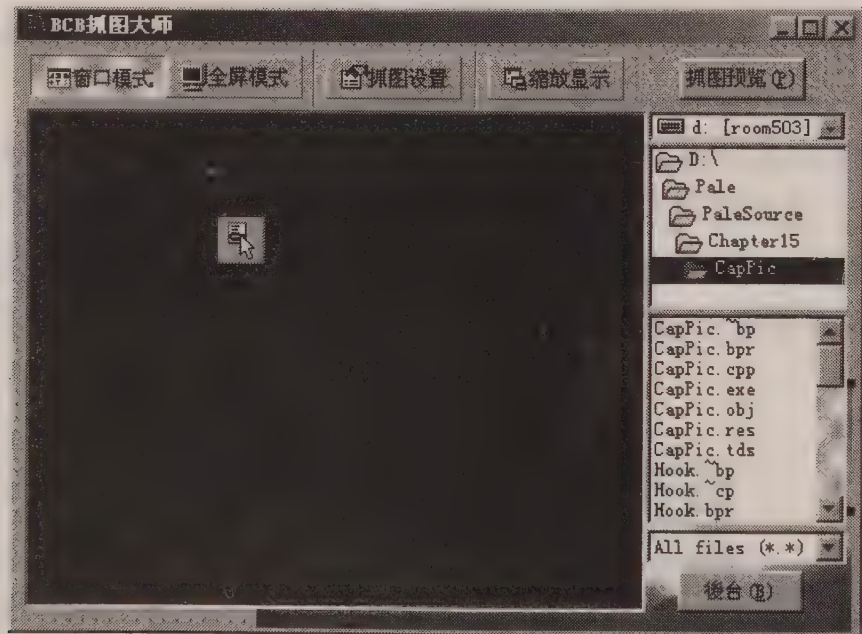


图 15-4　设计期间的范例程序主窗体

具体到“BCB 抓图大师”范例程序，程序总体分为两大部分：其一是主应用程序部分，这部分实现抓图程序界面，抓图选项设置和图像预览等功能，应用程序部分也负责完成调用抓图模块和与钩子函数通信；第二部分是一个 DLL 模块，这是抓图程序的核心部分，它完成

三项任务：

- 安装和卸载系统钩子。
- 钩子回调函数部分。在这部分中侦测用户按键并实现屏幕截图。
- 建立和操作内存映像文件的实现部分。

实际上，对于这三项任务可以放在不同的 DLL 中完成，但出于实例简洁的考虑，在 BCB 抓图大师程序中将它们揉合在同一个 DLL 中。

CapPic 工程中包含下面的一些文件：

- 主窗体单元。文件：Unit1.cpp、Unit1.h。
- 实现钩子函数调用和内存映像文件操作的动态链接库。文件：Hook.dll，该 DLL 由 Hook 工程编译得到。

另外，与上一章不同，在 BCB 抓图大师范例程序中，使用了 C++ Builder 5 新增的 TrayIcon 组件来实现程序任务栏指示区图标。TrayIcon 组件需要一个 ImageList 组件与之连接，该组件中包含在任务栏显示图像。通过响应 TrayIcon 组件的事件即可方便可靠的实现带有任务栏指示区图标的应用程序。例如本例中响应了 TrayIcon 组件的 OnMouseDown 事件，具体代码如下：

```
void__fastcall TForm1::TrayIcon1 MouseDown(TObjcet*Sender,
    TMouseButton Button,TShiftState Shift,int X, int Y)
{
    switch(Button){
    case mbLeft:
          Show();
          break;
    case mbRight:
          PopupMenu1->PopupComponent=Form1;
          SetForegroundWindow(Handle);
          PopupMenu1->Popup(X,Y);
          break;
    }
}
```

这段简单的程序实现了当用户鼠标左键点击图标显示程序，右键点击图标显示弹出菜单的功能。

15.4 钩子（Hook）函数

钩子（Hook）是 Windows 消息处理机制中的一环。在这里可以安装一个称为“钩子函数”的回调函数。钩子函数一旦被装进 Windows 的消息系统，它就可以在其他进程收到消息前来处理消息。通常，可以有各种类型的钩子被装入系统，来过滤并处理不同类型的消息。安装一个钩子很像在 DOS 系统中捕获一个中断。当用户安装钩子函数时，也许想要对消息作某些处理，然后调用钩子链（Hook Chain）的下一个钩子，以允许消息能够被系统滤出。

15.4.1 Windows 的钩子函数

Windows 编程和早期的 DOS 编程已有很大的不同，Windows 应用程序的编程必须遵循微软规定的一套“游戏规则”：建立窗口、为该窗口建立消息循环、在窗口函数完成各种工作。Windows 应用程序必须要遵守这个“定死”的框架，否则将不能正常执行。而且，在微软的“游戏规则”下，Windows 程序难以突破自身进程空间的限制。其实，Windows 编程中还是存在很多“后门”的，钩子函数就是其中的手段之一。

1. 钩子函数的安装和卸载

钩子函数技术是消息处理机制中留给应用程序的一个手段，应用程序可以利用它安装一个针对特定类型消息的子例程，使编程人员可以在某些消息到达目的地之前监视和修改它们。在 Win32 操作系统中，提供了 13 种类型的钩子函数。安装和卸载不同类型的钩子函数的方式是相同的。安装一个钩子函数必须使用 SetWindowsHookEx。该函数声明如下：

```
HHOOK SetWindowsHookEx(
  int idHook,              // 安装的钩子类型
  HOOKPROC lpfn,           // 钩子回调函数地址
  HINSTANCE hMod,          // 钩子函数所在程序的实例句柄
  DWORD dwThreadId         // 安装钩子的线程 ID
);
```

卸载钩子函数通过 Win32 API 函数 UnhookWindowsHookEx 完成。该函数需要指定卸下的钩子函数句柄：

```
BOOL UnhookWindowsHookEx (
  HHOOK hhk   // 卸载钩子函数句柄
);
```

这里有一个问题：对于某一类型的钩子，可能会存在多个应用程序为这型的钩子安装钩子函数，这时候系统就会形成一个钩子函数链（Hook Chain）。这时应用程序必须做一些工作使得钩子链能够传递下去，而不至于在创建的钩子函数中断掉（除非特殊情况不愿让别的钩子进行处理）。进行钩子链的下一个钩子需要调用 CallNextHookEx 函数：

```
LRESULT CallNextHookEx (
  HHOOK hhk,               // 当前钩子句柄
  int nCode,               // 钩子号
  WPARAM wParam,           // 传递给钩子函数的 wParam 值
  LPARAM lParam            // 传递给钩子函数的 lParam 值
);
```

2. Windows 钩子类型

Win32 定义了下列钩子类型：

- WH_CALLWNDPROC。

窗口过程钩子。在 SendMessage 函数被调用时，产生一个窗口过程的钩子调用，允许修改该消息。

- WH_CALLWNDPROCRET。

消息已在窗口过程处理后，接收这些消息的钩子。在 SendMessage 函数返回之后，产生窗口过程的钩子调用。

- WH_CBT。

基于计算机的钩子调用（CBT，computer-based training），发生在激活、创建、关闭、最小化、最大化、窗口移动和窗口尺寸改变之前，在完成一个系统命令之前，在清除一个鼠标和键盘事件之前，在设置输入焦点之前，以及在同步系统消息队列之前。

- WH_DEBUG。

针对钩子编程的调试钩子。在任何其他钩子被调用前调用。

- WH_FOREGROUNDIDLE。

前台空闲窗口钩子。当某个应用程序的前台线程进入空闲状态时调用。

- WH_GETMESSAGE。

接收投递消息的钩子。在 GetMessage 函数已搜寻到一个来自应用消息队列的消息时被调用。

- WH_JOURNALPLAYBACK。

回放以前输入消息钩子。用于将键盘和鼠标消息送入到系统消息队列中。

- WH_JOURNALRECORD。

录制输入消息钩子。调用发生在系统从系统消息队列中清除消息时。

- WH_KEYBOARD。

键盘钩子。调用发生在应用程序调用 GerMessage 或 PeekMessage 函数，并且存在键盘消息 WM_KEYDOWN 或 WM_KEYUP 等待处理时。

- WH_MOUSE。

鼠标钩子。调用发生在应用程序调用 GerMessage 或 PeekMessage 函数，并且存在某个鼠标消息等待处理时。

- WH_MSGFILTER。

应用程序消息钩子。调用发生在对话框、消息框、菜单已接收了一个消息后，但在该消息真正被处理之前。

- WH_SYSMSGFILTER。

系统消息钩子。调用发生在对话框、消息框、菜单已接收了一个消息后，但在该消息真正被处理之前。

- WH_SHELL。

外壳钩子。被外壳应用程序用来从系统接收通知消息。

15.4.2 使用钩子函数的问题

使用钩子函数和调用普通的 Windows API 有很大不同，在为进程或系统安装钩子函数时，程序指定了钩子回调函数用于处理钩子。但和一般的 API 函数回调函数不同，钩子函数

通常放在 DLL 中，使得它可以被系统和每一个进程访问，因此安装函数 SetWindowsHookEx 的第三个参数应该是 DLL 模块的句柄。

所以，为了安装钩子函数，必须首先创建一个 DLL，在该 DLL 中存放钩子回调函数。在钩子函数安装完成后，无论和钩子函数相关联的进程是否活动，钩子函数均会被调用。这样会降低系统的性能，甚至破坏系统稳定。因此使用钩子函数应该慎重，并且在钩子函数使用完毕后，应该立刻卸下。如果不正确的钩子函数锁住了系统，可以按下 Ctrl+Alt+Del 键，一般来说这样可以清除任何已安装的系统范围钩子。

对于不同类型的钩子，其作用的范围不同。钩子函数有两种作用范围：系统范围和线程范围，表 15-1 是各种类型钩子函数作用范围的说明。

表 15-1　不同类型钩子的作用范围

钩　子	作 用 范 围
WH_CALLWNDPROC	线程或系统
WH_CBT	线程或系统
WH_DEBUG	线程或系统
WH_FOREGROUNDIDLE	线程或系统
WH_GETMESSAGE	线程或系统
WH_JOURNALPLAYBACK	系统
WH_JOURNALRECORD	系统
WH_KEYBOARD	线程或系统
WH_MOUSE	线程或系统
WH_MSGFILTER	线程或系统
WH_SYSMSGFILTER	系统
WH_SHELL	线程或系统

对于支持线程范围的钩子类型，可以为某个特定的线程或进程安装钩子。钩子安装函数 SetWindowsHookEx 的第四个参数指定所要安装到的线程的 ID。如果需要安装系统范围的钩子，这个参数应置为 0。

线程范围的钩子回调函数是可以存在于应用程序中的（EXE 文件），而对于系统范围的钩子函数来说，就必须创建 DLL 为系统回调提供支持。

15.4.3　键盘钩子

现在回到本章要创建的实例——“BCB 抓图大师”。在本程序中将使用键盘钩子实现截图热键。思路很简单：为操作系统安装系统范围内的键盘钩子，一旦用户在任何时候按下键盘，钩子回调函数将被执行，在钩子函数中判断用户按键是否为设置的“热键”，如果与热键匹配，则进行抓图操作。

以下是在 Hook 工程中实现键盘钩子的相关代码：

Source　实现系统范围的键盘钩子

```
__declspec(dllexport)
void* __stdcall GetGMem();
```

```
__declspec(dllexport)
LRESULT __stdcall
KeyboardProc(int code,WPARAM wParam,LPARAM lParam);
__declspec(dllexport)
bool EnableKeyHook();
__declspec(dllexport)
bool DisableKeyHook();
HHOOK hNextHookProc;
void *pGMem;
HANDLE FileMapHandle;
//---------------------------------------------------------------------------
LRESULT __stdcall KeyboardProc(
    int code,WPARAM wParam,LPARAM lParam)
{
        int ss,sc,sa;
        if (code <0){
                CallNextHookEx(hNextHookProc, code, wParam, lParam);
                return 0;
        }
        if ((lParam&0x80000000)==0){
                TCanvas *Desk =new TCanvas();
                Graphics::TBitmap *Bmp=new Graphics::TBitmap();
                RECT rect;
                if ((GetKeyState(VK_CONTROL)&0x8000)==0) sc=0;
                else sc=1;
                if ((GetKeyState(VK_MENU)&0x8000)==0) sa=0;
                else sa=1;
                if ((GetKeyState(VK_SHIFT)&0x8000)==0) ss=0;
                else ss=1;
                if(wParam ==*(UINT *)pGMem&&
                *((BYTE *)pGMem+50)==sc&&
                *((BYTE *)pGMem+51)==sa&&
                *((BYTE *)pGMem+52)==ss)
                {       //判断按键是否与定义热键匹配
                if(*((BYTE *)pGMem+6)==0){
                        //窗口截图
                        HWND hWnd = GetForegroundWindow();
                        GetWindowRect(hWnd,&rect);
                        Desk->Handle=GetDC(0);
                        Bmp->Width=rect.right-rect.left;
```

```
        Bmp->Height=rect.bottom-rect.top;
        Bmp->Canvas->CopyRect(
            Rect(0,0,Bmp->Width,Bmp->Height),
            Desk,
            rect);
    }
    else if (*((BYTE *)pGMem+6)==1){
        // 全屏截图
        Desk->Handle=GetDC(0);
        rect=Rect(0,0,Screen->Width,Screen->Height);
        Bmp->Width=rect.right-rect.left;
        Bmp->Height=rect.bottom-rect.top;
        Bmp->Canvas->CopyRect(
            Rect(0,0,Bmp->Width,Bmp->Height),
            Desk,
            rect);
    }
    if (*((BYTE *)pGMem+7)==1)  // 截图到剪贴板
        Clipboard()->Assign(Bmp);
    AnsiString FileName;
    if (*((BYTE *)pGMem+5)==0){
        // 截图为 Bitmap 格式
        FileName=AnsiString((char *)pGMem+10)+
        IntToStr(*((BYTE *)pGMem+4))+ ".bmp";
        Bmp->SaveToFile(FileName);
    }
    else {
        // 截图为 JPEG 格式
        TJPEGImage *jpeg=new TJPEGImage();
        jpeg->Assign(Bmp);
        FileName=AnsiString((char *)pGMem+10)+
        IntToStr(*((BYTE *)pGMem+4))+ ".jpg";
        jpeg->SaveToFile(FileName);
        delete jpeg;
    }
    (*((BYTE *)pGMem+4))++;
    }
    else return 0;
    delete Bmp;
    delete Desk;
```

```
        }
        else {
            return 0;
        }
        return 1;
    }
    //---------------------------------------------------------------------------
    bool EnableKeyHook()
    {
        if (hNextHookProc != 0) return false;
        hNextHookProc = SetWindowsHookEx(WH_KEYBOARD,
            (HOOKPROC)KeyboardProc,
            HInstance,0);
        return (hNextHookProc != 0);
    }
    //---------------------------------------------------------------------------
    bool DisableKeyHook()
    {
        if (hNextHookProc !=0 )
        {
            UnhookWindowsHookEx(hNextHookProc);
            hNextHookProc = 0;
        }
        return (hNextHookProc == 0);
    }
    //---------------------------------------------------------------------------
    int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
    {
        hNextHookProc=0;
        return 1;
    }
    //---------------------------------------------------------------------------
```

这段程序的关键部分是 KeyboardProc 函数，这个函数是钩子被触发时的回调函数。可以看到，在调用 SetWindowsHookEx 安装钩子时引用了这个函数指针。在 KeyboardProc 函数代码中依次解决了下面几个问题：

- 当 code 值为 0 时，调用 API CallNextHookEx 继续钩子链。
- 判断用户按键是否为设置的热键，如果是则进行下面的截图工作。
- 实现多种方式的截图功能。

判断按键是否为设置的热键需要分别判断基本键和附加键是否匹配。对于基本键来说，可以认为就是 wParam，直接进行比较即可。对于附加键，我们需要使用 API 函数

GetKeyState，该函数可以返回当前某键的按键状态，其中返回值的最高位如为 1，表示该键正被按下，反之则表示该键未被按下。此处通过 GetKeyState 函数判断了 Ctrl、Alt、Shift 三键的按键状态。

具体到截图工作是较为容易实现的。在程序中我们通过 C++ Builder 的 TCanvas 类和全屏 DC 建立连接（使用 GetDC(0)获取全屏 DC），然后就可以通过 Canvas 对象进行处理和存储工作，这里涉及了一些 API 函数的调用，包括 GetForegroundWindow 和 GetWindowRect 函数。

在 KeyboardProc 函数中多次使用 pGMem 这个指针，这个指针为一块全局内存在进程映像的地址。在这块内存中存有主程序中抓图设置的信息，通过读取这些信息以决定采用哪种抓图方式抓图。关于这部分内容请参看后文的具体讲述。

15.5 进程间数据共享

钩子安装程序（主程序）和钩子函数实际存在于两个不同的进程（钩子函数安装后，与所关联的线程属同一进程空间），在 Win32 系统下分别享有两个不同的应用程序空间。然而我们需要将在主程序中一些用户设置信息传递给钩子函数执行部分。这涉及到进程间数据共享的问题。

15.5.1 利用内存映像文件共享数据

在 Win32 系统中有几种方法越过进程边界共享数据，但都没用通过 DLL，Win32 系统新的 DLL 机制使得我们不能像 Windows 3.X 中那样简单的通过 DLL 共享数据。使用 DLL 实现进程间数据共享的方法已经过时。但另一方面，在 Win32 系统的几种实现数据共享的代码在每个进程都要重复出现，所以可以把编写的实现数据共享代码放进一个 DLL 中。例如，我将讲述的一个 Win32 系统中常用的数据共享方法——使用内存映像文件分配全局存取内存，将会把它包括在 DLL 项目中供需要共享数据的多方应用调用。

内存映像文件（Memory Mapped File）技术允许编程人员创建可由系统页面文件（System Paging File）直接备份的文件映像对象，并可以指定需要多大的内存空间。同时会将内存映像文件映像到应用程序中，所以在应用程序中可以像操纵进程空间指针那样使用内存映像。

内存映像对象可操纵系统全局范围的系统页面文件。往其中写的数据可以被任意数目的应用程序存取，而且不要求遵守进程边界的限制。

Win32 API 中提供了一系列函数为这种技术提供支持。包括 CreateFileMapping、MapViewOfFile 和 UnmapViewOfFile，使用内存映像文件技术的具体方法如下：

1. 创建文件映像对象

创建文件映像对象需要调用 CreateFileMapping 函数，这个函数的声明如下所示：

```
HANDLE CreateFileMapping (
  HANDLE hFile,            // 文件映像句柄
  LPSECURITY_ATTRIBUTES lpFileMappingAttributes,      // 安全属性
```

```
    DWORD flProtect,          // 保护属性
    DWORD dwMaximumSizeHigh,        // 内存映像文件大小高位
    DWORD dwMaximumSizeLow,         // 内存映像文件大小高位
    LPCTSTR lpName            // 文件映像对象名称
);
```

hFile 参数指定与文件映像对象相关的文件名，一般来说我们会在内存中使用虚拟文件，这需要传递一个特殊的数值 0xFFFFFFFF。lpFileMappingAttributes 参数和 flProtect 参数分别用于传递安全和保护属性。安全属性可以使用缺省的安全属性，保护属性可设为 PAGE_READWRITE，使进程对创建的内存区域拥有读写权力。dwMaximumSizeHigh 和 dwMaximumSizeLow 参数合成为一个 64 位整数用以说明映像文件最大可以使用多少字节的空间。

lpName 为文件映像对象指定内部名称。可以用来从一个非初始调用 CreateFileMapping 的进程获得对文件映像的一个引用。例如，当 A 进程传递 GlobalMem 为文件映像对象名称，并调用 CreateFileMapping 函数；B 进程也传递 GlobalMem 的名称，调用 CreateFileMapping 函数。这样的结果是两个进程都可以存取这块内存区域，这就是如何实现数据共享的原因。

2. 将全局内存地址映像到进程空间

在文件映像对象被创建后，必须将划分的全局内存映像到进程地址空间才能够使用。这可以通过对 MapViewOfFile 函数来实现，该函数声明如下：

```
LPVOID MapViewOfFile (
    HANDLE hFileMappingObject,// 文件映像对象句柄
    DWORD dwDesiredAccess,      // 进入模式
    DWORD dwFileOffsetHigh,           // 偏移量高位
    DWORD dwFileOffsetLow,            // 偏移量低位
    DWORD dwNumberOfBytesToMap   // 指定映像到进程的字节数
);
```

参数 hFileMappingObject 是从 CreateFileMapping 函数返回的句柄，参数 dwDesiredAccess 用来设定存取模式，有 FILE_MAP_WRITE、FILE_MAP_READ、FILE_MAP_COPY、FILE_MAP_ALL_ACCESS 等值。参数 dwFileOffsetHigh 和 dwFileOffsetLow 参数组合成一个 64 位数值来指定映像开始的文件偏移位置，如果我们使用系统页面文件，应将这两个参数都设为 0。参数 dwNumberOfBytesToMap 说明映像到进程空间的字节数。如果将参数 dwNumberOfBytesToMap 传入 0，则 MapViewOfFile 会把 CreateFileMapping 调用中的参数 dwMaximumSizeHigh 和 dwMaximumSizeLow 指定的所有内存映像到进程空间。

3. 释放文件映像对象

在文件映像对象使用完毕后，应该解除文件映像对象，然后释放相关的文件映像对象句柄。利用 Win32 API 函数 UnmapViewOfFile 来解除一个文件的映像，该函数是这样声明的：

```
BOOL UnmapViewOfFile(
    LPCVOID lpBaseAddress        // 映像的起始地址
);
```

参数 lpBaseAddress 为 MapViewOfFile 函数返回的指针。在解除文件映像后，还应该释放文件映像对象句柄，可以通过 API 函数 CloseHandle 来完成。

15.5.2 在 DLL 中实现存取全局内容代码

现在，我们将上述技术应用到实际中去——创建 DLL 实现全局存取。在 BCB 抓图大师程序中，实现进程间通信的 DLL 与钩子函数所在的 DLL 是同一的，下面是在 Hook.dll 中增加的代码：

Source 使用内存映像文件共享数据

```
__declspec(dllexport)
void* __stdcall GetGMem();
void *pGMem;
HANDLE FileMapHandle;
//---------------------------------------------------------------------------
void UnmapMemory()
{
        if (pGMem){
                UnmapViewOfFile(pGMem);
                pGMem = NULL;
        }
        CloseHandle(FileMapHandle);
}
//---------------------------------------------------------------------------
void MapMemory(DWORD size)
{
        FileMapHandle = CreateFileMapping(
                (HANDLE)0xFFFFFFFF,NULL,PAGE_READWRITE,
                0,size,"_GMEM");
        if (FileMapHandle){
                pGMem = MapViewOfFile(FileMapHandle,
                        FILE_MAP_WRITE,0,0,0);
                if (pGMem==NULL){
                        UnmapMemory();
                        MessageBox(0,"映像文件失败!","错误",MB_OK);
                }
        }
        else MessageBox(0,"建立文件映像对象失败!","错误",MB_OK);
}
```

```
//---------------------------------------------------------------------------
void* __stdcall GetGMem()
{
        return pGMem;
}
//---------------------------------------------------------------------------
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
        hNextHookProc=0;
        switch (reason){
        case DLL_PROCESS_ATTACH:
                pGMem = NULL;
                FileMapHandle = NULL;
                MapMemory(100);
                break;
        case DLL_PROCESS_DETACH:
                UnmapMemory();
                break;
        }
        return 1;
}
```

这段程序中包含了内存映像文件处理的两个函数。MapMemory 函数创建内存映像文件对象，并将全局地址映像到进程空间。UnmapMemory 函数实现解除文件映像对象和释放文件映像对象句柄。在这里，DLL 的入口点函数 DllEntryPoint 很值得注意，入口点函数代码使用系统传递 reason 参数，在 DLL 第一次被进程装载时（reason 参数值为 DLL_PROCESS_ATTACH），进行初始化工作并调用 MapMemory 开辟 100 字节的全局存取空间，在 DLL 最终被卸载时（reason 参数值为 DLL_PROCESS_DETACH），调用 UnmapMemory 释放全局存取空间。可以看到，通过 DLL 的 reason 参数，非常适宜地解决了使用内存映像文件共享数据何时创建映像文件以及何时释放映像文件的关键问题。

在 DLL 中提供一个输出函数 GetGMem，这个函数简单地返回映像到进程空间的内存指针，在应用程序中获得这个指针就可以进行全局存取。当不同的应用调用 DLL 时，不同的应用程序通过内存映像文件就可进行进程间通信了。

15.6 截图程序的具体实现

前两小节已经解决了 BCB 抓图大师程序的两个关键性技术问题——钩子函数和进程间数据共享，并且完成了 Hook.dll 动态链接库的编制工作。现在回到真正的应用程序主体部分，该程序被实现为一个任务栏指示图标应用程序。主程序同时扮演下列三个角色：

- 键盘钩子函数的安装者。

- 截图设置信息向钩子函数的传递者。
- 一个可以浏览所截取图片和其他图片的浏览器。

基本上在解决了两个关键性的技术问题后，实现 BCB 抓图大师程序并不是一件太困难的事。现在需要处理的是一些细节的编程问题，包括设置（特别是热键设置）处理，钩子函数的安装和全局内存存储结构的数据组织。

15.6.1 全局存取内存区域的数据组织

前一节创建了通过内存映像文件实现存取全局内容代码。我们已经可以开辟一段可供不同进程访问的内存空间。现在，在 BCB 抓图大师程序中需要实际创建这样的全局存取的内存区域，并规定在这段内存区域中的数据信息存储结构。

在 Hook 工程（Hook.dll 动态链接库）中创建的 GetGMem 创建一段 100 个字节的内存区域并返回该区域在进程中的指针。在范例程序中，这样划分这段存储空间：

前 4 个字节　存储截图热键的基本键设定。

第 5 字节　图像文件编号。在本例中程序根据编号生成存盘图像文件名。请参看程序钩子函数中抓图实现部分。

第 6 字节　截图图像格式。规定 0 代表 Bitmap 格式，1 代表 JPEG 格式。

第 7 字节　抓图模式。0 代表窗口模式，1 代表全屏模式。

第 8 字节　是否抓图到剪贴板。

第 11—50 字节　存储截图路径字符串。

第 51—53 字节　存储热键附加键包括 Ctrl、Alt 和 Shift 键设定。

具体在主窗体的 FormCreate 函数中这样初始化数据：

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    EnableKeyHook();
    pBuf = GetGMem();
    strcpy((char *)pBuf+10,"D:\\");
    // 默认热键 Ctrl+F
    *(UINT *)pBuf=Word('F');
    *((BYTE *)pBuf+50)=1;
    *((BYTE *)pBuf+51)=0;
    *((BYTE *)pBuf+52)=0;
    *((BYTE *)pBuf+4)=0; // 图片起始编号
    *((BYTE *)pBuf+5)=0; // 截图为 Bitmap 格式
    *((BYTE *)pBuf+6)=0; // 窗口模式截图
    *((BYTE *)pBuf+7)=0; // 不拷贝到剪贴板
    PathName = AnsiString((char *)pBuf+10);
    HotKey=Word('F');
    FilterComboBox1->Filter=
```

```
        "Bitmap files (*.Bmp)|*.bmp|JPEG files(*.jpg)|*.jpg";
    FileListBox1->Mask=FilterComboBox1->Mask;
    TrayMessage(NIM_ADD);
}
```

在全局存取的内存区域中填写的数据将会被 DLL 中的钩子函数读取，而后钩子函数的截图部分会根据设置信息的不同采取不同的截图方式。具体请参看 15.4 节中列出的钩子函数代码。

另外，主窗体中提供“窗口模式/全屏模式”的切换选择，简单的读写全局存取的内存区域就可以完成，以下是两个按钮响应函数的代码：

```
void __fastcall TForm1::SpeedButton1Click(TObject *Sender)
{
    *((BYTE *)pBuf+6)=0;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::SpeedButton2Click(TObject *Sender)
{
    *((BYTE *)pBuf+6)=1;
}
//---------------------------------------------------------------------------
```

下面是主窗体中其他功能的代码，包括截图图像浏览的实现：

```
void __fastcall TForm1::SpeedButton3Click(TObject *Sender)
{
    // 缩放显示切换
    if (SpeedButton3->Down)
        Image2->Stretch=true;
    else
        Image2->Stretch=false;
}
void __fastcall TForm1::FilterComboBox1Change(TObject *Sender)
{
    // FileListBox 的 Mask 选择
    FileListBox1->Mask=FilterComboBox1->Mask;
}
void __fastcall TForm1::FileListBox1Click(TObject *Sender)
{
    // 预览图像文件
    Image2->Picture->LoadFromFile(FileListBox1->FileName);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
        // 预览上次截图图像
        if (*((BYTE *)pBuf+4)>0){
        if (*((BYTE *)pBuf+5)==0){
              Image2->Picture->LoadFromFile(PathName+
                    IntToStr(*((BYTE *)pBuf+4)-1)+".bmp");
        }
        else {
              Image2->Picture->LoadFromFile(PathName+
                    IntToStr(*((BYTE *)pBuf+4)-1)+".jpg");
        }
        }
}
//---------------------------------------------------------------------------
```

15.6.2 抓图设置处理

除了“窗口模式/全屏模式”的选择之外，其他的抓图设置选项通过“截图设置”对话框实现，如图 15-5 所示。

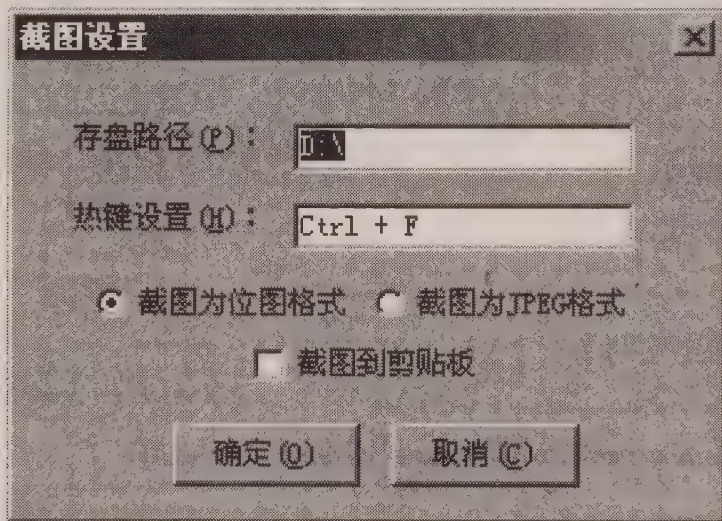


图 15-5　BCB 抓图大师程序的截图设置对话框

截图设置窗体中较为特殊的是热键设置的处理。在实现热键设置功能时我们使用了 C++ Builder 提供的 THotKey 组件。

THotKey 组件是 Windows 热键通用控件的一个封装，THotKey 组件的外观和单行文本编辑组件 TEdit 颇为相似（图 15-5），但当 THotKey 组件成为焦点时，用户所按下的键将会在编辑框中显示出来，同时热键信息将作为 THotKey 组件的属性存储下来。

THotKey 组件的关键属性包括 HotKey、InvalidKey 和 Modifiers。

- HotKey 属性

声明：_property Menus::TShortCut HotKey = {read=GetHotKey, write=SetHotKey, nodefault};

包含 HotKey 组件的当前热键信息。为 TShotCut 类型。

- Modifiers 属性

声明：__property THKModifiers Modifiers ={read=FModifiers,write=SetModifiers, nodefault};

指定 THotKey 组件的默认附加键，当用户按下无附加键的热键时，Modifiers 属性所指定的附加键将自动被加上。例如，Modifiers 属性之值为 hkShift，则当用户按下 A 键时，实际热键为 Shift+A。

- InvalidKey 属性

声明：__property THKInvalidKeys InvalidKeys = {read=FInvalidKeys, write=SetInvalidKeys, nodefault};

该属性允许编程人员指定一个或多个附加键不可被设置。可取值为：hcNone，hcShift，hcCtrl，hcAlt，hcShiftCtrl，hcShiftAlt，hcCtrlAlt，hcShiftCtrlAlt。

用户在截图设置窗体输入的信息，在主窗体中写入全局存取内存，具体代码如下：

```
void __fastcall TForm1::SpeedButton4Click(TObject *Sender)
{
    if(Form2->ShowModal()==ID_OK){
        ShortCutToKey(Form2->HotKey1->HotKey,HotKey,ShiftState);
        *(UINT *)pBuf=HotKey;
        if (ShiftState.Contains(ssCtrl))
            *((BYTE *)pBuf+50)=1;
        else
            *((BYTE *)pBuf+50)=0;
        if (ShiftState.Contains(ssAlt))
            *((BYTE *)pBuf+51)=1;
        else
            *((BYTE *)pBuf+51)=0;
        if (ShiftState.Contains(ssShift))
            *((BYTE *)pBuf+52)=1;
        else
            *((BYTE *)pBuf+52)=0;
        PathName=Form2->Edit1->Text;
        strcpy((char *)pBuf+10,PathName.c_str());
        if (Form2->CheckBox1->Checked)
            *((BYTE *)pBuf+7)=1;
        else
            *((BYTE *)pBuf+7)=0;
        if (Form2->RadioButton1->Checked)
            *((BYTE *)pBuf+5)=0;
        else
            *((BYTE *)pBuf+5)=1;
```

```
        }
        SpeedButton4->Down=false;
    }
```

这里需要特别注意的是热键信息的处理和存储。THotKey 组件的 HotKey 属性存储热键信息为 TShortCut 类型，我们必须将 TShortCut 类型转换为我们可以理解的热键信息。完成这个任务需要使用 C++ Builder 中的 ShortCutToKey 函数，该函数声明如下：

```
void __fastcall ShortCutToKey (
  TShortCut ShortCut,        // 需要解析的 TShortCut 类型变量
  Word &Key,                 // 基本键虚拟键值
  Classes::TShiftState &Shift    // 附加键
);
```

通过 ShortCutToKey 函数获得我们容易理解的热键信息，然后就可以将热键信息以我们规定的方式存储在内存映像文件的相应位置。这样，钩子函数 DLL 中的代码就可以获得这些信息，并实现截图功能。

第16章

COM对象、自动化和XCoolClock控件
——组件对象模型（COM）

COM 接口及实现原理
创建 COM 对象服务器
IDispatch、双重接口和 dispinterface
OLE Automation
类型库（Type Library）
ActiveX 技术和创建 ActiveX 控件



本章导读

组件对象模型（Component Object Model，简称 COM）是组件对象之间相互接口的规范。它提供了为保证能够互相操作，客户和组件应遵循的一些标准。在 COM 架构下可以将各种各样功能专一的软件组件通过接口“搭接”起来，构成复杂的应用系统。这样带来的好处是多方面的。

以目前的情势来看，COM 毫无疑问是将来软件开发的方向，本章将深入讨论 COM 相关的各种编程问题。然而组件对象模型是一个相当大的话题，所以本章中将提供多个应用实例，以演示 COM 技术的各个方面。包括编写可以供多种编程语言使用的代码（COM 对象），自动化 Word 及 Excel 应用程序，利用 COM 接口实现更为强大的 BCB WebBrowser 程序的第三版本，以及创建 ActiveX 控件 CoolClockX。

本章内容包括:

- 理解 COM 接口及 COM 实现原理。
- 如何建立服务器程序以实现 COM 对象。
- 通过接口调用 COM 对象，IDispatch、dispinterface 和双重接口。
- 通过 OLE Automation 控制其他应用程序。使用类型库（Type Library），ActiveX 技术和创建 ActiveX 控件。

COM 技术是相当复杂的，并且有很多概念并不容易理解。如果您属于对 COM 了解较少的读者，则应该多花些时间思考书中提到的关于 COM 的问题。并且，在本章中会出现一些有意思和有意义的实例，这些例子可以帮助您较容易地掌握强大的 COM 技术。

16.1 理解 COM 接口及其实现

组件对象模型 COM 是组件对象之间相互接口的规范，接口在 COM 技术中是一个非常重要的概念。所谓接口，实际上就是一组属性和方法，通过调用这些属性和方法，外部代码就可以访问对象，并且不需要知道对象具体是怎样实现的。但本质上，接口本身并不实现自己，接口的方法都是虚方法。真正实现接口的是另外一个类，称之为接口对象。

16.1.1 关于 COM 基本概念

软件重用一直是人们追求的目标，开发人员希望能够象搭积木一样随意“装配”应用程序。软件重用的关键问题在于各个程序模块之间的互相通信。然而，程序模块间的交换信息并没有一种标准的方法，虽然像 DLL 导出函数、动态数据交换（DDE）、Windows 剪贴板和 Windows API 本身，以及其他一些过去的标准，例如 VBX 和 OLE1 都可以进行程序模块间相互通信，但这些是不够的。我们所需要的是一种广泛的、通用的、稳固的方案，它必须适用于各种各样的程序模块的通信方式。现在，Microsoft 的组件对象模型（COM）正是这样一个标准。

老的标准总有这样那样的问题，这些标准往往是单一领域的，并且有的在性能方面有较大问题（例如 VBX 不能在 32 位环境下使用）。而 COM 则为 Windows 提供了统一的、可扩充的、面向对象的通信协议。COM 技术具有以下几点优势：

- 为 Win32 客户应用程序载入和调用 Win32 DLL 提供一种标准的、与语言无关的方法。
- 为一个 EXE 和同一机器上的另一个 EXE 通信提供通用的方法（替代 DDE）。
- 用 ActiveX 控件代替 VBX。
- 为应用程序与操作系统交互操作提供一种新的方法（例如 Win32 Shell API）。
- 提供适应新协议的良好可扩充性。可以不断的通过加入新的接口扩充 COM 功能，而不会影响以前版本的运行。
- DCOM（Distribute COM，分布式组件对象模型）允许不同机器上的程序模块相互通信，即使两台机器上使用了不同的微处理器。

1. COM 的本质

COM 到底是什么？这个问题并不容易回答。下面援引 Microsoft 的一段描述来回答这个问题：

组件对象模型（COM）是 OLE 的根基，它同时提供程序设计的模型和 OLE 组件的二进

制规范。COM 定义并且实现了软件组件的机制，例如应用程序、数据对象、控件和服务程序都可成为相互影响的对象。一个软件对象包含一些数据体，连同一个或多个提供数据访问和使用的函数。一个 OLE 组件实现进入对象数据通过一套或多套专用的相互关联的函数称为接口（interfaces）。（引自 MSDN Library）

Microsoft 给出的 COM 定义包含了很多内容：首先，COM 是个二进制规范，它与源代码无关，所以即使 COM 对象是由完全不同的编程语言创建的，即使运行在不同的进程空间和不同的操作系统平台，这些程序模块（ COM 对象）仍可以毫无阻碍的互相交换信息；其次，COM 既定义了规范，同时也实现这些规范，它以 COM 库的形式提供了访问 COM 对象核心功能的标准接口以及一组 API 函数，这些 API 用于实现创建和管理 COM 对象的功能。

这个定义提到了组件对象概念，组件对象与一般意义上的对象有所区别，一般意义上的对象是将数据和对象方法封装在一起的数据类型的实例，对象方法是可以直接调用的。而组件对象只能使用接口（Interface）而不是方法提供服务。

2. 什么是 COM 接口

COM 接口的实质是一个纯虚的基类（在 C++中往往是一个结构，因为 C++的结构具有可继承性），它定义了一组纯虚函数，这些函数完全控制 COM 对象行为的一个方面。接口的精确定义为：基于对象的一组语义上相关的功能。

真正实现接口的是接口对象（Interface Object），接口对象继承于其对应的接口。一个 COM 对象可以拥有超过一个的接口，每一个接口代表操纵对象的一个方面。实际上可以理解为，COM 模型中的对象方法是分组的，每一组对应了一个 COM 接口。

接口是客户程序与 COM 对象通信的唯一途径，客户程序不能直接访问 COM 对象中的实际数据，而仅能通过接口访问接口对象中的方法。如果一个对象拥有多个接口，COM 技术允许客户程序调用 COM 对象的标准成员函数 QueryInterface 查询组件对象所支持的其他接口。当客户程序调用 QueryInterface 函数后，如果组件对象确实支持要求的接口，函数就返回该接口的指针。如果组件对象不支持该接口，QueryInterface 则返回出错信息。这种策略提供客户程序动态了解 COM 对象所支持接口的能力。

在 COM 技术中，接口是一个极为重要的概念，接口机制是 COM 中的关键技术。可以将接口理解为两个软件模块之间达成的协议，但是，采用接口到底有什么好处呢？接口机制的优势基本上可以归纳为两点：其一，接口提供了一种统一的应用程序通信方法。在 COM 技术中，无论 COM 对象所在的服务器是一个 DLL、EXE 还是一个控件，与之通信和操作的方式都是一致的；其二，无论 COM 对象是否进行更新或升级，接口的定义是不变的。如果想改进 COM 对象，加入新的功能，都不改变原来定义的接口，而是增加新的接口。例如，DirectX 中具有一个 IDirectDraw 接口，随着 DirectX 的不断更新，在 DirectX 6.X 版本中已经拥有了 IDirectDraw、IDirectDraw2、IDirectDraw3、IDirectDraw4 四个接口，可以使用 IDirectDraw4 接口利用新的 DirectX 特性，然而，原先使用 IDirectDraw 接口的程序依然可以正确执行。

仅仅讲述理论是不容易深刻理解的，下面本书将引入在 DLL 中创建对象的问题讲述 COM 的实现原理，读者可以从中深刻的理解 COM 接口的含义。

16.1.2 在 DLL 实现类

前面两章讨论了有关 DLL 的各种问题。一般来说，DLL 是函数的集合。但是，在 DLL 中也可以实现类，并且在与 DLL 链接的程序中使用该类的对象。实际上，我们会看到 DLL 中的类就是 COM 对象的雏形。

DLL 规范本身并不是面向对象的，DLL 中只能导出函数而不能导出一个类。所以，DLL 中不存在导出类，它导出的是该类的虚方法。

1. 在 DLL 实现类

在 DLL 中实现类应该具备建立对象、并向调用它的应用程序返回对象的能力。同时，DLL 返回对象时，还应该返回纯虚对象方法所占的内存地址。

如何具体实现 DLL 中的类定义？首先，应该建立需要导出的抽象类的定义，另外，在 DLL 中还要建立一个对应的实现类，最后还需要建立一个导出函数，该函数建立一个实际的对象。

下面是 DLL 中的具体代码。

Source 编写 DLL 中的类

```
//---------------------------------------------------------------------------
class TDllNum
{
public:
      virtual int GetValue()=0;
      virtual void SetValue(int Value)=0;
      virtual void Inc()=0;
      virtual void ShowMessage()=0;
};
//---------------------------------------------------------------------------
class TDllNumImpl: public TDllNum
{
private:
      int FValue;
public:
      int GetValue();
      void SetValue(int Value);
      void Inc();
      void ShowMessage();
      TDllNumImpl();
};
```

```
//---------------------------------------------------------------------------
TDllNumImpl::TDllNumImpl()
{
        FValue=0;
}
//---------------------------------------------------------------------------
int TDllNumImpl::GetValue()
{
        return FValue;
}
//---------------------------------------------------------------------------
void TDllNumImpl::SetValue(int Value)
{
        FValue=Value;
}
//---------------------------------------------------------------------------
void TDllNumImpl::Inc()
{
        FValue++;
}
//---------------------------------------------------------------------------
void TDllNumImpl::ShowMessage()
{
        MessageBox(0,"Hello Component Object Model!",
                "DllObject",MB_OK);
}
//---------------------------------------------------------------------------
extern "C" __declspec(dllexport)  void __stdcall NewObj(void** ppObj);
void __stdcall NewObj(void** ppObj)
{
        TDllNumImpl* pObj=new TDllNumImpl();
        *ppObj=pObj;
}
```

这段代码中演示了建立一个简单的类。其中，声明一个该类的抽象类 TdllNum，在该类的声明中，这个类的所有对象方法必须都声明为 virtual，因为只有纯虚方法才可以通过 VTable（虚函数表，也称为 vtbl）从 DLL 中调出。

这段程序中另外建立了一个实现类 TDllNumImpl，该类是 TDllNum 的继承类，它对应于 TDllNum 类实现了对象方法，并且加入了一个实的构造函数，该类用于创建实际的对象。

最后，程序中建立 DLL 的导出函数 NewObj 实现对象实例的创建。

2. 使用 DLL 中的类

上面已经完成了在 DLL 中类的实现。接下来可以在应用程序中使用在 DLL 中所定义的类。但使用 DLL 类与使用普通类的方法稍有不同，在引用 DLL 的应用程序中，还必须做一些工作使 DLL 中的类得以正确调用。

在这样的客户应用程序中，关键是需要再次建立一个带有纯虚方法的类的声明，即声明一个实际意义上的接口。声明这个类时，关键是所有对象方法具有相同的参数，并且按照相同的顺序排列，而不需要方法名称与在 DLL 中的声明保持一致。也就是说，在新的声明中可以改变对象方法的名称，而不会影响该对象方法的实际工作。例如可以如下声明：

```
class TDllNumber
{
public:
        virtual int GetValue();
        virtual void SetValue(int Num);
        virtual void Increase();
        virtual void Message();
};
```

可以看到，这里的声明和在 DLL 中类的声明有所不同，SetValue 方法的参数名改为了 Num，并且对象方法 Inc 改为了 Increase。但这些改动完全不会影响程序。这是因为，在 DLL 中和客户程序中的类的声明，关键在于虚方法的定义顺序、参数类型，而不在于它的名称。实际上，类的方法是通过 VTable 调用的，C++ Builder 在调用时仅仅会注意是调用第一个方法，还是第二个方法，而不会注意对象方法的实际名称。所以，在应用程序中重声明 DLL 中的类时，关键是要注意类方法顺序的一致。

在声明类之后，就可以使用导出函数 NewObj 生成一个类的实例。

```
NewObj((void **)&pNumObj);
```

对于这样的实际对象，它会包含一个指向对象数据的指针，而对象数据的第一个元素就是指向它所在类的 VTable 的指针。指针的关系可以由图 16-1 表示。

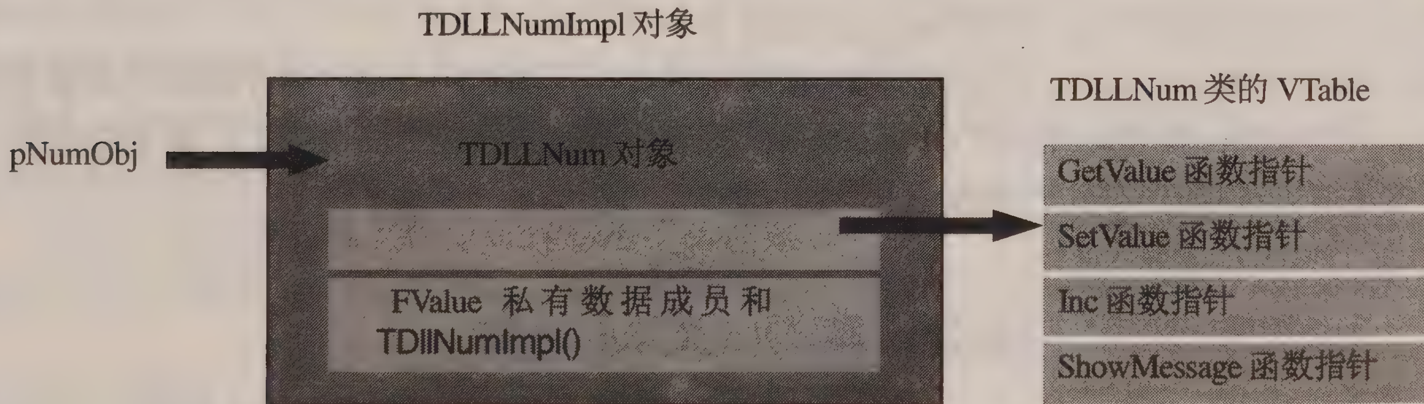


图 16-1　构建和使用 DLL 中的类：指针关系

再下来就可以通过 pNumObj 指针使用这个对象。操作这个对象如同使用普通的对象一样，在此不再赘述。这个程序运行结果如图 16-2 所示。

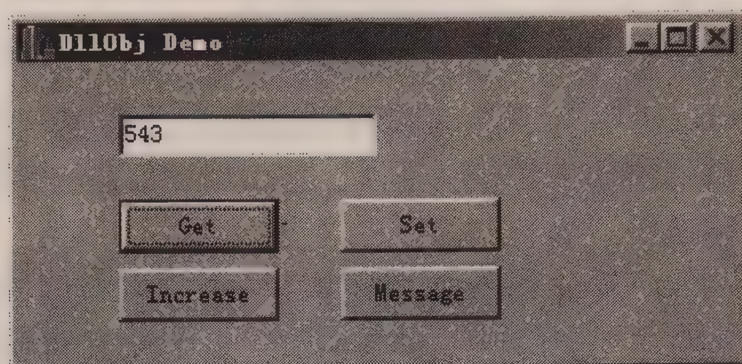


图 16-2　使用 DLL 中的类的客户程序

3. 分析 DLL 中的类

前面提到，DLL 中类的实现其实就是一个 COM 对象的雏形，实际上，对上面建立的 TDllNum 类进行进一步的改造后，就可以成为一个标准的 COM 对象。虽然这些改造工作可能并不是非常简单。在实现 DLL 类的过程中，我们做了一些很关键的工作，实际上，这些工作可以和创建标准的 COM 对象的一些重要概念对应起来。

首先，在 DLL 工程中定义了一个纯虚类 TDllNum，这个纯虚类实际上是一个 COM 对象的接口，只不过一般建立接口时采用 struct 声明而非 class，例如：

```
truct TDllNum
{
public:
    virtual int GetValue()=0;
    virtual void SetValue(int Value)=0;
    virtual void Inc()=0;
    virtual void ShowMessage()=0;
};
```

而 TDllNum 的实现类 TDllNumImpl 实际上可以认为是一个接口对象，只有实现了接口对象，接口才可以真正被访问，TDllNumImpl 完成了一些实际的工作。

输出函数 NewObj 实际上在扮演 COM 对象中的类工厂的作用，通过这个函数才能够真正获得一个类的实例。COM 中的类工厂正是完成构造对象的工作。当然，类工厂要复杂的多，并且它也是通过接口（COM 标准接口 IClassFactory）调用的。

16.2　实现 COM 对象

这一小节将在 C++ Builder 5 中实际完成一个 COM 对象。由于使用了 C++ Builder 5 作为 COM 的开发工具，创建 COM 对象服务器和使用 COM 对象的繁琐操作在一定程度上被简化了，C++ Builder 会自动完成许多事情，使得程序开发人员可以把精力集中到实际起作用的代码上来。

C++ Builder 5 使用了 Microsoft 的 ATL（ActiveX Templet Library，ActiveX 模板库）对创建 COM 对象提供支持。Visual C++也同样使用 ATL，但在 C++ Builder 中开发 COM 依然要比 Visual C++简单，这是因为 C++ Builder 提供了类型库编辑器 Type Library Editor，可以方便快捷有效的创建、编写和管理对象，同时 C++ Builder 可以自动生成很多框架代码，包括 ATL 不直接支持的双重接口（dual interface）等。

ATL 本身是一套非常复杂的库函数，它简化了 COM 的实现。而 C++ Builder 在 ATL 的基础上又做了许多工作，使得 COM 开发进一步简化。本章将不涉及 ATL 的具体工作机制问题。

16.2.1 COM 的服务程序类型

COM 提供的是一种统一的程序模块间的相互通信方案，但具体的 COM 程序模块的类型却是多种多样的。可以认为，COM 技术采用的是客户/服务器模式，COM 对象在某个服务程序模块中，而客户程序则可以通过接口调用服务程序中的 COM 对象。

如何得到某个 COM 服务器的组件对象的服务？客户程序首先要请求 Windows 去查找 COM 对象的位置，找到以后把接口的指针传递给客户。而 Windows 则是通过系统注册表查找并得到 COM 组件所在服务器的位置。也就是说，COM 服务程序必须在 Windows 的注册表中注册。

COM 服务程序是 COM 对象的容器，COM 服务程序的形式是多种多样的。总的来说，根据 COM 服务与 COM 客户各自运行的进程地址空间关系，COM 服务程序分为三类：In-Process 服务器、Local 服务器和 Remote 服务器。In-Process 服务器通常是 DLL，它映射到客户的进程地址空间中运行，例如 ActiveX 控件可以认为是一种 In-Process 服务器。Local 服务器通常是 EXE，有它自己的进程空间，但与 COM 客户在同一个机器上，例如一些以 COM 标准建立的独立应用程序，如 Microsoft Word、Internet Explorer、AutoCAD 等。Remote 服务器可以是 EXE 也可以是 DLL，它运行在与 COM 客户程序不同的机器上。实现 Remote 服务器的技术被称为 DCOM（Distribute COM，分布式组件对象模型），它是 COM 的一个扩展。图 16-3 是 In-Process 类型服务器的结构图。

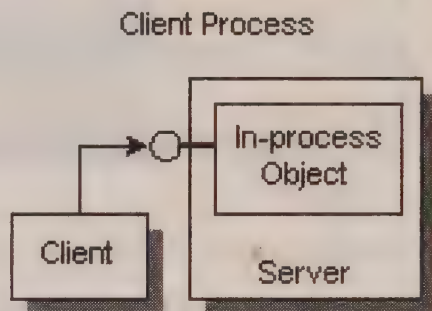


图 16-3 In-Process 类型的服务器

另外，服务器和客户程序也可以集成在同一程序中，这样的例子很多，例如 Microsoft Word、Microsoft Excel 等。实际上，它们是一种包含了客户应用的 COM 服务器。

16.2.2 创建 COM 对象

这一小节将着手创建一个简单的 COM 对象。我们使用 C++ Builder 完成这项任务，这就意味着将可以绕过许多复杂艰深的工作。然而，为了使读者能够较为深入地理解 COM 对象的实现，本节将分析 C++ Builder 自动完成的代码。

本小节中所创建的 COM 对象服务程序是 In-Process 类型的，也就是说，它将是一个 DLL。实际上，这个 COM 对象和在第 1 节创建的 DLL 中的类——TDllNum 完成同样的工作。

为创建 In-Process 类型的 COM 服务器，选择【**File/New**】菜单，在 ActiveX 页选择 ActiveX Library 选项，开始 COM 服务器程序的开发。本节的范例程序项目被命名为 COMObj。

下面是所生成代码的关键部分：

```
//---------------------------------------------------------------------------
STDAPI __export DllCanUnloadNow(void)
{
    return (_Module.GetLockCount()==0) ? S_OK : S_FALSE;
}
STDAPI __export DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    return _Module.GetClassObject(rclsid, riid, ppv);
}
STDAPI __export DllRegisterServer(void)
{
    return _Module.RegisterServer(TRUE);
}
STDAPI __export DllUnregisterServer(void)
{
    return _Module.UnregisterServer();
}
//---------------------------------------------------------------------------
```

这是四个由系统自动生成的输出函数，这组函数是 COM 需要并且由系统调用的。其中：

- DllCanUnloadNow 函数检查服务器是否输出了其所有对象并可以从内存下载。
- DllGetClassObject 函数提供一个标准函数用于访问 COM 对象。
- DllRegisterServer 和 DllUnregisterServer 函数实现在 Windows 系统注册表中添加和删除与服务器相关的信息。

这四个函数已经有了缺省实现，一般情况无须改动它们。现在，我们就有了一个完成的 In-Process 服务器框架。以下将继续通过 C++ Builder 的 COM 向导在这个服务程序中添加实际的 COM 对象。

选择【**File/New**】菜单，在 ActiveX 页选择 COM Object 选项，会弹出一个向导对话框，该对话框如图 16-4 所示，在这里，需要输入组件对象类名（CoClass）、线程模式（Threading Model）以及说明。C++ Builder 会自动根据组件对象类名生成接口名称。

选择 OK 后，C++ Builder 就会在服务程序中添加完整的 COM 对象框架。同时，类型库编辑器（Type Library Editor）自动被打开，如图 16-5 所示。下面就可以通过类型库编辑器向 COM 对象添加属性和方法，如图 16-6 所示。

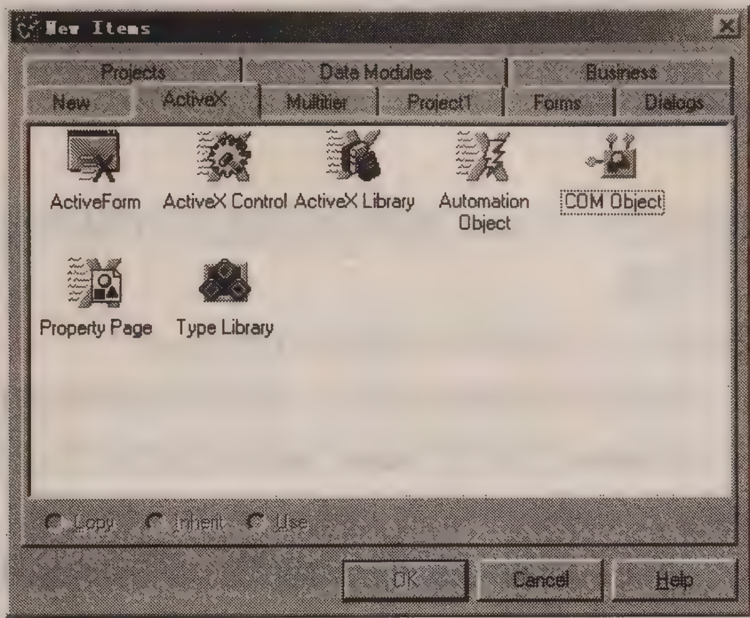


图 16-4　往 COM 服务器中添加一个 COM 对象

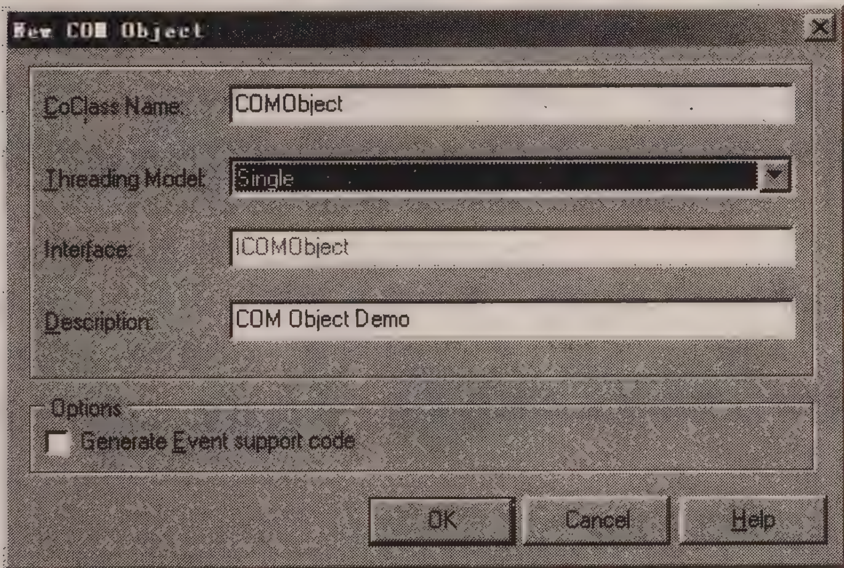


图 16-5　COM 对象向导对话框

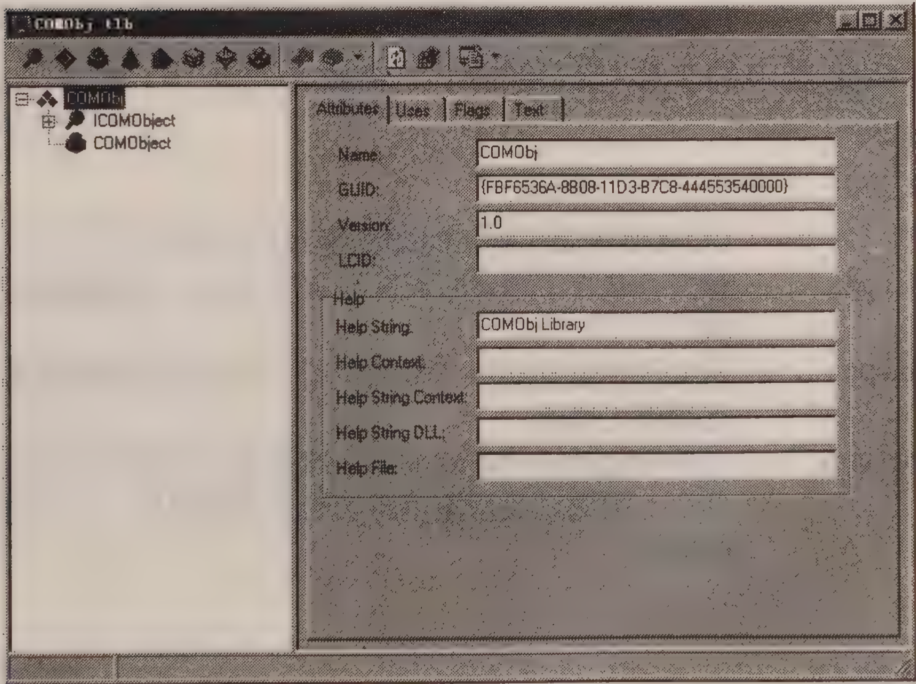


图 16-6　使用类型库编辑器构建 COM 对象

1. 为 COM 对象添加属性和方法

现在我们来看看如何使用 C++ Builder 类型库编辑器为 COM 对象添加实际功能，向 COM 对象添加方法应该依照以下步骤：

(1) 选择要添加方法的接口，在本例中选择 ICOMObject。

(2) 单击编辑器上方工具栏中的 Method 按钮。类型库编辑器将自动生成一个缺省名称，如 Method1。在 Attributes 页面的 Name 编辑框修改该方法名称，或者在编辑器左边的树视图中直接修改。在本例中设为 Inc。

(3) 切换到 Parameters 页面，编辑对象方法的参数，单击 Add 按钮可以加入为该方法添加一个参数。可以修改参数名（Name）、参数类型（Type）、以及传递标志（Flags）。在本例中无须添加参数。

(4) 在 Parameters 页面的 Return Type 字段修改对象方法的返回类型。在本例中也无须修改。

(5) 最后可以在 Flags 页面做一些调整，并在 Text 页面检查生成的 IDL 代码。

(6) 单击工具栏的 Refresh 按钮，将 COM 对象的编辑接口反映到 CPP 源码中。



IDL（Interface Definition Language，接口定义语言）是一个工业标准。它是一种脚本语言，通常用来定义对象。实际上，不仅仅是 COM 对象，CORBA 对象也是使用 IDL 定义的。在 C++ Builder 中，我们可以在类型库编辑器查看并修改 IDL 代码，并且所做的修改会影响到类型库编辑器的其他内容。

COM 不直接支持属性，COM 对象的属性必须由两个相应的 set 方法和 get 方法支持。接下来我们创建一个 Value 属性，需要单击 Property 按钮，在类型库编辑器中将会出现 Value 属性的 get 和 set 方法。类似的可以调整并刷新，之后，再向 COM 对象添加 ShowMessage 方法，方法同上。

此时，在类型库编辑器中的工作已经完成了。在整个过程中，C++ Builder 自动生成了许多重要文件，说明如下：

COMObj.cpp	COM 服务器代码。DLL 的主入口文件。
COMObj_ATL.cpp	该文件包含了项目中所需的 ATL 代码。不需要修改该文件的内容。
COMObj_TLB.cpp	该文件中包含接口和类的说明，可以通过类型库编辑器编辑，无须手工修改。
COMObjectImpl.cpp	该单元包括了接口的实现。这是唯一需要手工修改的文件，需要在上面通过类型库编辑器给生成的属性、方法框架中加入实际代码。

2. 完成接口的实现

在创建的简单 COM 对象中，实现接口对象是很容易的。打开 COMObjectImpl.h 文件，找

到 TCOMObjectImpl 类的声明，在 public 字段前加入私有成员 FValue：

```
private:
    long FValue;
```

实现 TCOMObjectImpl 类的构造函数：

```
TCOMObjectImpl()
{
    FValue=0;
}
```

打开 COMObjectImpl.cpp 文件，加入实现接口对象的属性方法的具体代码。

```
STDMETHODIMP TCOMObjectImpl::get_Value(long* Value)
{
    try
    {
        *Value=FValue;
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_ICOMObject);
    }
    return S_OK;
};
STDMETHODIMP TCOMObjectImpl::Inc()
{
    try
    {
        FValue++;
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_ICOMObject);
    }
    return S_OK;
};
STDMETHODIMP TCOMObjectImpl::set_Value(long Value)
{
    try
    {
        FValue=Value;
    }
    catch(Exception &e)
```

```
        {
            return Error(e.Message.c_str(), IID_ICOMObject);
        }
        return S_OK;
};
STDMETHODIMP TCOMObjectImpl::ShowMessage()
{
        try
        {
            MessageBox(0,"Hello Component Object Model!",
                "COMObj Demo",MB_OK);
        }
        catch(Exception &e)
        {
            return Error(e.Message.c_str(), IID_ICOMObject);
        }
        return S_OK;
};
```

这样就完成了一个完整的 COM 对象。现在，可以编译工程，并选择【**Run/Register ActiveX Server**】注册 COM 服务器。另外，也可以使用 Microsoft 的 RegSvr32.EXE 工具安装 COM 服务器，可以在 Windows\System 目录找到这个程序。

16.2.3 创建客户程序

这一小节将继续创建一个客户程序，以演示如何在应用程序中使用标准的 COM 对象。客户程序 Test 被放在一个独立的目录中，因为 COM 服务器已经在系统注册表中注册，所以，客户程序无须知道 COM 服务器驻留的目录就可以调用它。

该程序的窗体界面与测试 DLL 中的类的窗体界面是相同的，但作为 COM 对象，必须通过特殊的方式调用。

首先，必须引入注册的 COM 服务器的类型库。选择菜单【**Project/Import Type Library**】，在对话框的列表框中找到 Project1 Library (Version 1.0)一项，单击 OK 按钮，C++ Builder 将会引入上面创建的 COM 对象的类型库，并自动生成 COMObj_TLB.cpp 和 COMObj_TLB.h 文件。COMObj_TLB 单元是 C++ Builder 分析了 COM 服务器的类型库（.tlb 文件或.olb 文件）自动生成的。其中包含了 TCOMICOMObjectT 模板类。

由于有了 COM 对象的类型库，下面可以通过优化智能接口（smart interface）来调用 COM 对象。这是一种简单有效的办法。

在窗体头文件中包含 COMObj_TLB.h 头文件，并为 TForm1 声明私有成员：

```
TCOMICOMObject COMObj;
```

下面是在该程序中使用 COM 对象的代码：

Source 在客户端程序使用 COM 对象

```
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
        COMObj=CoCOMObject::Create();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        Edit1->Text=AnsiString(COMObj->get_Value());
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        COMObj->set_Value(StrToInt(Edit1->Text));
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
        COMObj->Inc();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
        COMObj->ShowMessage();
}
//---------------------------------------------------------------------------
```

调用 COM 对象首先必须获得一个接口对象的实例。实际上，程序使用 CoClassCreator 代理生成对象实例。对象 CoCOMObject 通过 CoClassCreator 的模板类定义，它是由 C++ Builder 自动生成的。对于 CoClassCreator 的机制稍后还要解释，以下是 CoCOMObject 类的声明：

```
typedef TCoClassCreatorT<TCOMICOMObject, ICOMObject, &CLSID_COMObject, &IID_ICOMObject> CoCOMObject;
```

CoCOMObject 具有 Create 方法，调用这个函数的结果返回 TCOMICOMObject 对象实例。CoCOMObject 还有一个 CreateRemote 方法，该函数可用于在 DCOM 中构建远程对象。

接下来就可以通过伪指针语法直接调用该对象的方法。称之为伪指针语法是因为我们获得的是一个类的静态实例，而非通常意义的动态实例。虽然调用的方式非常像指针对象，但

使用->进行调用仅仅是因为智能接口重载了这个操作符。

除了智能接口，还可使用标准 COM 方式调用组件对象。定义一个 ICOMObject 类型的指针变量，并使用 CreateComObject 函数创建 COM 对象。

16.3 几个关键问题

上一节中，我们已经完成了一个完整的 COM 对象及服务器，但实际上，由于 C++ Builder 在幕后做了大量的工作，使我们没有看到一些关键性的东西，导致我们对 COM 的实现内幕还不是很清楚。

本节将研究 C++ Builder 生成的代码，深入 COM 机制，讨论几个 COM 技术中的关键问题。这些内容对深刻了解 COM 是非常有帮助的。

16.3.1 GUID、CLSID 和 IID

在创建 COM 对象时生成的 COMObj_TLB.cpp 文件中可以看到如下的代码：

```
namespace Comobj_tlb
{
extern "C" const GUID LIBID_COMObj = {0xFBF6536A, 0x8B08, 0x11D3,{ 0xB7, 0xC8, 0x44, 0x45, 0x53, 0x54, 0x00, 0x00} };
extern "C" const GUID IID_ICOMObject = {0xFBF6536B, 0x8B08, 0x11D3,{ 0xB7, 0xC8, 0x44, 0x45, 0x53, 0x54, 0x00, 0x00} };
extern "C" const GUID CLSID_COMObject = {0xFBF6536D, 0x8B08, 0x11D3,{ 0xB7, 0xC8, 0x44, 0x45, 0x53, 0x54, 0x00, 0x00} };
};
```

这段代码的作用是声明组件对象的几个 GUID。那么，什么是 GUID，它有怎样的作用呢？

在 Windows 系统中，存在着大量的各种形式的 COM 对象，每个 COM 对象有可能拥有大量接口。GUID 就是用来标识这些对象和接口的。GUID 是 Globally Unique Identifier 的缩写，意为全局唯一标识符，它的严格定义是：操作系统或程序用来引用 COM 对象的唯一数字。

在 COM 对象中包含一些 GUID 的现实形式，如 LIBID、IID、CLSID。当 COM 服务器注册后，Window 注册表就会存储这些 ID。C++ Builder 定义了 GUID 类型的结构，该结构是这样声明的：

```
typedef struct  _GUID
{
  DWORD Data1;
  WORD Data2;
  WORD Data3;
  BYTE Data4[ 8 ];
} GUID;
```

GUID 类型是一个相当长的数字，它总长达 16 个字节或 128 位。统计理论告诉我们这样

的数字已经“不可能”出现两个同值的随机数。但却不能使用随机数来填充一个 GUID，因为不是所有的数字都是有效的，Windows 会检测 GUID，无效的 GUID 将不被识别并出现错误信息。正确的方法是使用 API 函数 CoCreateGuid 生成有效的 GUID。事实上，C++ Builder IDE 集成了这个 API 调用的功能，当在代码编辑器的任何地方按下 Ctrl+Shift+G 键，将会自动出现有效的 GUID 序列。

如果使用 C++ Builder 的 COM 向导创建组件对象，则没有必要手工生成 GUID，所有需要 GUID 的地方 C++ Builder 都会自动完成。GUID 有几种现实形式，包括 LIBID（Library ID），标识服务程序类型库；CLSID（Class ID），标识服务器中特定的组件对象；IID（Interface ID），标识组件对象的接口。

16.3.2 IUnknown 接口

在上一节创建 COM 对象的代码中，注意到接口的定义方式：

```
interface ICOMObject : public IUnknown
{    ……
```

可以看到一个名为 IUnknown 的东西。COM 技术中，IUnknown 是一个特殊的接口，IUnknown 是所有接口的基类。

IUnknown 接口定义了三个标准的 COM 接口对象方法：QueryInterface、AddRef 和 Release。由于所有的 COM 接口都继承于 IUnknown，所以每一个 COM 接口的 VTable 的前三个函数都是 QueryInterface、AddRef 和 Release，这使得所有的 COM 接口都可以当作 IUnknown 接口来处理。

IUnknown 中包含名为 QueryInterface 的对象方法，QueryInterface 是一个非常重要的函数，客户端可以通过此函数来查询某个组件是否可支持特定的接口。

QueryInterface 带有两个参数：

```
HRESULT __stdcall QueryInterface(const GUID &IID, void **pObj);
```

其中第一个参数 IID 通过 GUID 标识一个用户所需的接口，另外一个参数 pObj 用于返回 QueryInterface 请求的接口指针的地址。

IUnknown 的另外两个方法与引用计数有关。引用计数是一种机制，其工作原理是：当接口对象第一次创建时，引用计数的初始值为 1；当再有一个客户请求获得接口对象的指针时，就调用 AddRef，使该计数加 1；当某客户不再需要组件对象的服务时，它应当调用 Release。注意，Release 并不真正释放接口对象，因为可能还有其他客户正在使用，Release 只是使引用计数减 1。只有当引用计数正好减为零时，接口对象才被删除。

使用 Borland C++ Builder 4 创建 COM 对象时，C++Builder 会自动提供接口的 QueryInterface、AddRef 和 Release 三个方法的缺省实现。然而，实际上这三个方法的实现并不复杂，例如，可以编写一个这样的 AddRef 函数：

```
int TInterface1::AddRef(void)
{
    FRefCount++;
    return FRefCount;
```

}

同样，QueryInterface 和 Release 的实现也不困难。QueryInterface 可以通过编写一个 switch 结构判断 IID，并返回相应的接口指针。在 C++ Builder 中，还有一种更为简单的方法，就是通过 TObject 类的 GetInterface 对象方法，可以直接获得一个指定 IID 的接口指针。

16.3.3 类工厂（ClassFactory）

除了 IUnknown 接口之外，COM 对象还有一个必须具备的特殊接口，这就是名为 IClassFactory 的类工厂接口。该接口对应实际的类工厂类。所谓类工厂（ClassFactory），更为准确地说，应该称为“对象工厂”，它用于构造对象实例。与所有的接口一样，IClassFactory 也是从 IUnknown 派生而来的。IClassFactory 的主要成员函数是 CreateInstance，定义方式如下：

```
HRESULT __stdcall
CreateInstance ( IUnknown* pUnknownOuter, const GUID &IID, void ** ppvObj );
```

在 C++ Builder 中，一般也无须考虑类工厂的问题，因为 C++ Builder 已经出色地实现了类工厂（虽然我们没有看到具体代码）。同时，C++ Builder 向导还会创建一个组件类生成器类（通过 TCoClassCreatorT 模板类），该类实现了 Create 和 CreateRemote 方法，可以方便地通过这个特殊的类获得组件对象的智能接口实例。

16.4 IDispatch、双重接口及 dispinterface

前面研究了基本的 COM 对象的创建和实现。其实，还有其他多种类型的 COM 对象，例如 OLE 自动化对象和 ActiveX 控件等。这一小节将讨论 OLE 自动化对象的一些问题。

16.4.1 创建 Automation 对象

有了创建基本 COM 对象的经验，创建 OLE 自动化对象不再是一件难事。本节将创建 EXE 形式的服务器，即一个 Local 服务器。

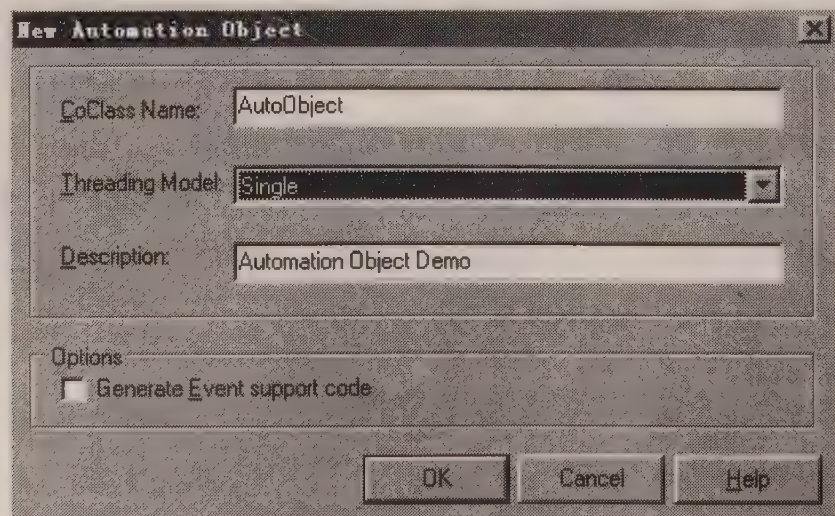


图 16-7 通过向导创建一个 Automation 对象

建立一个新的 Application，然后选择【File/New】菜单，然后在 New Items 对话框中选择 ActiveX 页，单击 Automation Object 图标。这时会出现 Automation Object Wizard 窗口，提示输入类的名称、线程模式等，按图 16-7 所示填写。

存盘并命名工程后，就获得了一个服务器主程序、一个用来定义 Automation 对象的附加模板类源文件和一个类型库。也就是说，一个完整的 Automation 服务程序框架已经建立了。以下的工作和创建 COM 对象时基本相同——通过类型库编辑器，向 Automation 对象加入一个可读写的属性 Value，一个 Inc 方法和一个 ShowMessage 方法。刷新，然后在 AutoObjectImpl.cpp 文件中加入属性和方法的实现代码。

Source 实现 Automation 对象的定义接口

```
STDMETHODIMP TAutoObjectImpl::get_Value(long* Value)
{
    try
    {
        *Value=FValue;
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_IAutoObject);
    }
    return S_OK;
};
STDMETHODIMP TAutoObjectImpl::Inc()
{
    FValue++;
    return S_OK;
}
STDMETHODIMP TAutoObjectImpl::set_Value(long Value)
{
    try
    {
        FValue=Value;
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_IAutoObject);
    }
    return S_OK;
```

```
};
STDMETHODIMP TAutoObjectImpl::ShowMessage()
{
        MessageBox(0,"Hello Auotmation Object!",
                "AuotObj Demo", MB_OK);
        return S_OK;
}
```

1. Automation 对象的实质

自动化（Automation）也被称为 OLE 自动化，它是客户控制组件对象的另外一种方法。在 16.2 的例子中，客户程序和 COM 对象的通信都是通过 COM 接口直接完成的，而自动化方法是一种非直接通过 COM 接口的访问对象方法，它提供另一种通信方式使得解释性语言和宏语言访问 COM 对象更为容易。

自动化不是独立于 COM 的，它是建立在 COM 的基础上的。一个自动化对象其实是一个实现了 IDispatch 特殊接口的 COM 对象，通过 IDispatch 接口的成员函数可以实现对自动化组件对象中函数的间接调用。

自动化关注的是运行时的类型检查，自动化技术可以不通过引入类型库而直接访问 COM 对象。但这是以牺牲速度和编译时的类型检查为代价的。实现了自动化的 COM 对象对于宏语言的使用更为容易，但对于组件对象的开发者——C++语言来说，则需要做更多的工作。

自动化目前的应用已相当普遍，主要是为宏语言功能提供支持，例如 Microsoft Word 和 Microsoft Excel 的自动化，以及 VBScript 和 JavaScript 用于 Web 页面的 ActiveX 控件的自动化。

2. 代码分析

在 AutoObj_TLB.h 头文件中，可以看到如下一行代码：

```
interface IAutoObject : public IDispatch
{    ……
```

可以看到 IAutoObject 接口被声明为继承于 IDispatch。IDispath 类实现了 IDispatch 接口。回忆以下在 16.2 节中创建普通 COM 对象时，相应的声明代码为：

```
interface ICOMObject : public IUnknown
{    ……
```

普通组件对象并没有实现 IDispatch 接口。

IDispatch 接口是一个非常特殊的接口，它可以是对象被不支持虚拟函数表（VTable）的语言，如 Visual Basic，VFP 等语言调用。但使用 IDispatch 接口会带来一些不必要的开销，从而降低速度。关于这个问题稍后我们还将详细讨论。

在 AutoObjectImpl.h 文件中可以找到 C++ Builder 生成的如下代码。

Source AutoObject 对象的声明

```
class ATL_NO_VTABLE TAutoObjectImpl :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<TAutoObjectImpl, &CLSID_AutoObject>,
  public IDispatchImpl<IAutoObject, &IID_IAutoObject, &LIBID_AutoObj>
{
private:
        long FValue;              // 此行为手工加入
public:
        TAutoObjectImpl()
        {
                FValue=0; // 此行为手工加入
        }
        // 与注册对象相关代码
        DECLARE_THREADING_MODEL(otSingle);
        DECLARE_PROGID("AutoObj.AutoObject");
        DECLARE_DESCRIPTION("Automation Object Demo");
        static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
        {
                TTypedComServerRegistrarT<TAutoObjectImpl>
                regObj(GetObjectCLSID(), GetProgID(), GetDescription());
                return regObj.UpdateRegistry(bRegister);
        }
        // 导出接口
        BEGIN_COM_MAP(TAutoObjectImpl)
                COM_INTERFACE_ENTRY(IAutoObject)
                COM_INTERFACE_ENTRY(IDispatch)
        END_COM_MAP()
        // 属性和方法的说明
public:
        STDMETHOD(get_Value(long* Value));
        STDMETHOD(Inc());
        STDMETHOD(set_Value(long Value));
        STDMETHOD(ShowMessage());
};
```

这段代码为 TAutoObjectImpl 类的声明。声明 TAutoObjectImpl 类使用了 ATL_NO_VTABLE 宏，ATL_NO_VTABLE 是一个可以帮助编译器产生高效代码的优化器。这个宏在 Include\Alt\ATLBase.h 头文件中定义。

可以看到 TAutoObjectImpl 类是多重继承的，继承 IDispatchImpl<IAutoObject, &IID_IAutoObject,&LIBID_AutoObj>保证类将会自动实现 IDispatch。这里没有必要深入了解 IDispatchImpl 模板类的工作原理，实际上，它的功能隐含实现 IDispath::GetIDsOfName 和 IDispath::Invoke 的复杂工作。

接下来是加入的一些成员变量和构造函数的代码，再下来的一组代码与组件对象的注册相关：

```
DECLARE_THREADING_MODEL(otSingle);
DECLARE_PROGID("AutoObj.AutoObject");
DECLARE_DESCRIPTION("Automation Object Demo");
```

这里需要关注的是对象的 Prog_ID，它被插入注册表中。在本例中，Prog_ID 为“AutoObj.AutoObject”，一些宏语言可以通过 Prog_ID 完成自动化对象的建立。

实际导出接口的工作由下面的代码完成：

```
BEGIN_COM_MAP(TAutoObjectImpl)
    COM_INTERFACE_ENTRY(IAutoObject)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

这里导出了 IAutoObject 接口和 IDispatch 接口，对于自动化对象，不仅可以使用 IDispatch 接口，也可以像通常的 COM 对象那样使用实际接口访问。

最后，可以看到组件对象的属性和方法的说明：

```
public:
    STDMETHOD(get_Value(long* Value));
    STDMETHOD(Inc());
    STDMETHOD(set_Value(long Value));
    STDMETHOD(ShowMessage());
```

对于 COM 对象方法的声明，C++ Builder 统一采用了 STDMETHOD 宏，这个宏的展开可以在 ObjBase.h 头文件中看到，实际上，这个宏是把函数声明为__stdcall 并且声明返回值为 HRESULT。

16.4.2 创建调用 Automation 对象的客户程序

客户程序能够访问驻留于服务器上的自动化对象。如前所述，自动化对象是特殊的 COM 对象，所以，可以使用标准 COM 对象调用方式或智能接口创建和操作一个自动化对象。但由于自动化对象实现了特殊接口——IDispatch，我们获得了更多不同的方法访问一个自动化对象，一种方法允许编程人员通过 Variant::Exec 的方式操作对象，另一种技术与 dispinterface 调度接口相关。

图 16-8 是测试 Automation 对象的客户程序，使用了三种方法访问对象。

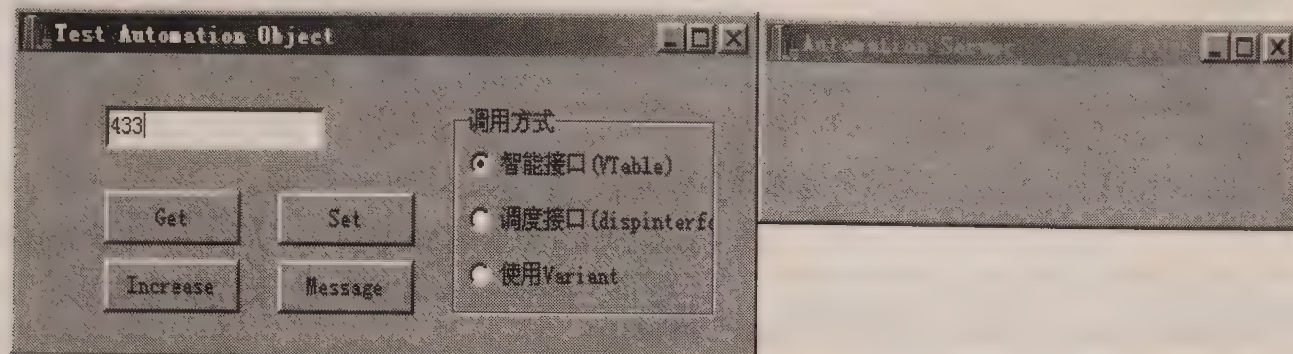


图 16-8　测试 Automation 对象的客户程序

Test2 程序演示使用上述三种不同的方法调用 Automation 对象。由于对象服务器为 Out-of-process 类型服务器，Test2 程序运行时，服务器程序会被自动启动，另一方面，尽管创建的 Automation 对象和 16.2 节完成的 COM 对象功能相同，但由于服务器性质不同，本例中 ShowMessage 对象方法将是在服务程序中弹出对话框而非客户程序。

在客户程序中，依然需要引入对象类型库。并且为主窗体类增加下面三个成员：

```
IAutoObjectDisp IAutoObj;          // dispinterface method of binding
TCOMIAutoObject AutoObj;           // VTable method of binding
Variant Auto;                      // Variant
```

以下是使用这三种技术操纵自动化对象的具体程序代码，但没有列出 Inc 方法的调用。

Source 使用不同的方式操纵自动化对象

```
//---------------------------------------------------------------------------

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
      IAutoObj.BindDefault();
      AutoObj = CoAutoObject::Create();
      Auto=CreateOleObject("AutoObj.AutoObject");
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Button4Click(TObject *Sender)
{
      switch (RadioGroup1->ItemIndex){
      case 0:
            AutoObj->ShowMessage();
            break;
      case 1:
            IAutoObj.ShowMessage();
            break;
```

```
    case 2:
        Auto.Exec(Procedure("ShowMessage"));
        break;
    }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    switch (RadioGroup1->ItemIndex){
    case 0:
        Edit1->Text=AnsiString(AutoObj->get_Value());
        break;
    case 1:
        Edit1->Text=AnsiString(IAutoObj.Value);
        break;
    case 2:
        Edit1->Text=Auto.Exec(PropertyGet("Value"));
        break;
    }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    switch (RadioGroup1->ItemIndex){
    case 0:
        AutoObj->set_Value(StrToInt(Edit1->Text));
        break;
    case 1:
        IAutoObj.Value=StrToInt(Edit1->Text);
        break;
    case 2:
        Auto.Exec(PropertySet("Value")<<StrToInt(Edit1->Text));
        break;
    }
}
//---------------------------------------------------------------------------
```

1. 通过 dispinterface 访问对象

在大多数情况下，C++程序员会使用优化的智能接口来调用自动化服务器，例如在本书 16.2 节中所示。这种方法体现了通过 COM 对象真实接口进行操作的方式。但是，一些诸如

Visual Basic 的工具对于直接通过真实接口访问并不支持。

Borland C++ Builder 支持另一接口访问自动化对象，该接口被称为 dispinterface。dispinterface 是在 C++Builder 创建自动化服务器类型库时自动获得的。dispinterface 是一个具有通过 IDispatch 来实现对象调度 ID 全部信息的接口。具体到本例中，在引入类型库时 C++Builder 会生成名为 IAutoObjectDisp 的 dispinterface。

在程序清单中我们看到了使用 dispinterface 访问 Automation 对象的方法，使用 dispinterface 在初始化时应使用 BindDefault 方法连接服务器。

dispinterface 的实现机制是怎样的呢？实际上，dispinterface 介于 IDispatch 接口和真正的标准 COM 接口之间。对于 IDispatch 接口来说，它允许编程人员查询一个对象并要求一系列可以用来访问对象方法和属性的数字标记，这种数字标记被称为调度标识符（DispID）。用于进行这项查询工作的方法是 IDispatch.GetIDsOfNames。

当获得了对象的某个属性或方法的 DispID 后，就可以通过 IDispatch 来调用该方法的实际函数或属性。完成这项工作的调用是 IDispatch.Invoke。

可以看到，当 IDispatch 要访问某个实际方法或属性时，必须要进行两次调用，首先获得调用标记，然后才可进行实际访问。而 dispinterface 则不同，它已经知道所需要的 DispID，所以可以单程访问一个 IDispatch 接口（不再是通常的“交谈”过程）。通过 dispinterface 的访问最终归结为 IDispatch.Invoke 的调用，但在幕后调用 IDispatch.Invoke 前并不需要进行查询，所以较之通常意义的 IDispatch 接口速度大大增加了，但它仍没有使用真实接口那样快，因为 dispinterface 依然是一种间接调用方式。

2. 通过 Variant 访问对象

如果不能访问头文件或类型库的话，仍然可以使用 IDispatch 接口访问自动化对象。其具体步骤和访问在上文有所涉及，实际上是调用 IDispatch.GetIDsOfNames 和 IDispatch. Invoke 的过程。但是在 C++ Builder 中提供一个使用相同机制并更为简单的方法，即使用 Variant 类型及 Variant::Exec。

在上面的程序清单中可以看到使用 Variant 变量访问自动化对象的示例。在使用这种技术时首先要通过 CreateOleObject 函数创建一个对象的实例，该函数需要传递想要创建对象的 ProgID。ProgID 是一个由服务器名称、一个小数点间隔和自动化对象名称组成的字符串。ProgID 在自动化对象注册时被写入系统注册表，在注册表的 HKEY_CLASSES_ROOT 部分可以看到系统中所有注册的自动化服务器的清单，并可以进一步查找自动化对象的 ProgID。CreateOleObject 函数返回包含 IDispatch 实例的 Variant。

使用 Variant 类型访问自动化对象是简便的，因为它支持幕后的 IDispatch. Invoke 和 IDispatch.GetIDsOfNames 的自动调用，我们仅需要传递想调用的方法名称。具体来说，是使用 Variant 类的 Exec 方法。例如：

```
Excel.Exec(PropertyGet("Charts")).Exec(Function("Add"));
```

在 C++ Builder 中使用 Variant 调用对象属性和方法在写法上不如其他语言中简单，例如在 Delphi、Visual Basic、JavaScript 中可以直接通过“.”来引用对象的属性或方法，上面一行代码如果在 Delphi 中将是：

```
Excel.Charts.Add;
```

C++ Builder 在使用 Variant 调用时的复杂性是 C++的严格语法所致，但无论是 C++、

Delphi 或 VB，在这种方式下访问自动化对象的实质是相同的。

在 C++ Builder 中，一般情况不会使用 Variant 来访问自动化对象，因为它在所有自动化对象的访问方法中是最慢的。但使用 Variant 的确存在有利之处，它允许在类型库或接口声明不存在时访问对象，同时，在属性和方法的参数传递方面也更为简便和灵活，在稍后的内容中还会接触到使用 Variant 访问对象的例子。

16.4.3 IDispatch 和双重接口

现在，回过头来讨论一些 OLE 自动化技术中更为深入的内容，即 IDispatch 接口和 C++ Builder 的双重接口机制。

1. IDispatch

在前面已经多次提到 IDispatch 接口，IDispatch 接口在 OLE 自动化中具有举足轻重的地位。这个接口最初是为 Visual Basic 或宏语言访问 COM 对象而设立的，但 C++程序员也可以访问它。

在 IDispatch 中最令人感兴趣的是两个成员函数 GetIDsOfNames 和 Invoke。对于 OLE 自动化对象来说，并不需要事先知道它的类型库，因为 GetIDsOfNames 函数能够在运行期间把名称转换为调度标识符（DispID）。这就意味着，在编译期间，编译器并不进行类型匹配检查，这样有可能导致使用 Invoke 函数调用失败。IDispatch 的 Invoke 函数是加在成员身上的外套，它的第一个参数就是真正要访问的成员函数调度标识符。同时，Invoke 函数传递的参数是不固定的，因为想要调用的函数的参数是不固定的。Invoke 函数是这样被声明的：

HRESULT __stdcall Invoke(DISPID dispIdMember, REFIID riid, LCID lcid, WORD wFlags, DISPPARAMS FAR* pDispParams, VARIANT FAR* pVarResult, EXCEPINFO FAR* pExcepInfo, unsigned int FAR* puArgErr);

在 C++ Builder 中，通常没有必要直接调用 GetIDsOfNames 和 Invoke，因为 C++ Builder 提供更为简洁有效的方式，无论使用 dispinterface 还是 Variant，关于 IDispatch 接口的调用都在幕后自动完成了。

2. 双重接口

在 OLE 自动化对象操作中，有的客户程序希望直接通过 VTable 来访问自动化服务程序，而有的客户程序则希望只在运行期通过 IDispatch 接口来访问自动化对象，例如，Visual Basic 等解释型语言或脚本工具，它们只在运行期才是有意义的。为了同时支持上述两种访问方式，C++ Builder 提供双重接口机制来解决这一问题。

双重接口（Dual Interface），是指一种能同时支持虚拟函数表（VTable）和运行期间 IDispatch 访问接口，双重接口同时以 VTable 和 IDispatch 接口两种方式暴露它的方法。在默认情况下，C++Builder 为所有对象创建双重接口。

其实双重接口并不像想象中那样复杂。实际上，它是具有虚函数表的接口，类似一个真实接口。但是双重接口除了包含 IUnknown 接口中的虚拟方法和自动化对象本身的方法之外，还包含了 IDispatch 接口的四个方法。这样，双重接口既可以简单的进入 VTable 调用方法，也

可以支持 IDispatch.Invoke 和 IDispatch 的其余功能。

图 16-9 形象的显示了双重接口的 VTable 结构。在名为 IMyInterface 的双重接口对应的 VTable 中，头三个是 IUnknown 接口中的虚拟方法_AddRef、_Release 和 QueryInterface，接下来就是 IDispatch 接口中的四个方法，然后才是自动化对象本身的方法。

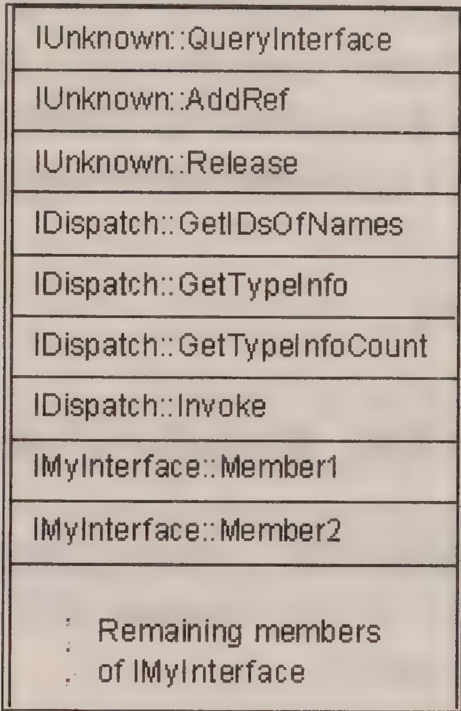


图 16-9 双重接口的 VTable 结构

具体到本节创建的 OLE 自动化对象 AutoObject，其双重接口的声明是这样的：

```
interface IAutoObject : public IDispatch
{
public:
    virtual HRESULT STDMETHODCALLTYPE get_Value(long* Value/*[out,retval]*/) = 0; // [1]
    virtual HRESULT STDMETHODCALLTYPE set_Value(long Value/*[in]*/) = 0; // [1]
    virtual HRESULT STDMETHODCALLTYPE Inc(void) = 0; // [2]
    virtual HRESULT STDMETHODCALLTYPE ShowMessage(void) = 0; // [3]
    ......
};
```

双重接口具有一些明显的好处，它具备了直接由 IUnknown 而来的真实接口的大多数优点，而且这一类型的类仍然可以为其他编程语言所获取，因为它同时实现 IDispatch。

16.5 实现 Word 和 Excel 自动化

前一节已经建立了 OLE 自动化连接的客户机端和服务器端，并讨论了自动化技术的很多问题。如果我们的目的是让两个应用程序合作运行，自动化确实是非常有效的，但其他的一些技术也可以实现进程间的通信，如在第 15 章讲到的内存映像文件以及简单的使用剪贴板。然而，OLE 自动化技术的真正价值是，它是一种标准（普遍意义下基于 COM 标准），可以使用它将 C++程序和用户拥有的其他应用程序集成在一起。典型的例子就是与 Microsoft Office

应用程序的集成，或与像 AutoDesk AutoCAD 这样独立的应用程序的集成。因为这些著名的程序本身都是标准的自动化服务器。

使用 OLE 标准，程序员可以通过 C++程序控制这些应用程序的方方面面，这种做法至少有以下两个优点：

- 利用这些应用程序容易实现在 C++程序中不易完成工作。例如，使用它们可以轻松处理数据格式，从数据库数据中生成报表与备注（利用 Excel）。
- 将 C++程序强大的数据计算和处理能力与应用程序融合，从而扩大这些应用程序的功效，同时可以建立对应于这些应用程序的可重用的处理程序。

16.5.1 使用 Variant 进行自动化

前一节中接触了使用 Variant（变体）类型变量访问自动化对象，本小节将更为深入讨论变体类型，因为它与自动化相关。

Variant 在 C++ Builder 中以类的形式实现，Variant 可以称为是 C++ Builder 中最为复杂的数据类型，它的核心数据格式由 16 字节的 TVarData 结构定义，该结构的声明在头文件 Include\vcl\systvar.h 中，声明的一部分如下：

```
struct TVarData
{
 union
 {
  Word      VType;   // Delphi-compatible - Variant Type member
  VARTYPE   vt;      // tagVARIANT compatible member
 };
 Word Reserved1;
 Word Reserved2;
 Word Reserved3;
 union
 {
  // Delphi-compatible Variant members
  Smallint    VSmallint;   // iVal
  Integer     VInteger;    // lVal
  Single      VSingle;     // fltVal
  ……
  // tagVARIANT compatible members (from OAIDL.H)
  // Allowing all types marked as [V] (may appear in a VARIANT to be initialized
  LONG       lVal;
  BYTE       bVal;
  SHORT      iVal;
  FLOAT      fltVal;
```

```
        DOUBLE      dblVal;
        VARIANT_BOOL boolVal;
        SCODE       scode;
        CY          cyVal;
        DATE        date;
        BSTR        bstrVal;
        IUnknown    *punkVal;
        IDispatch   *pdispVal;
        ……
    };
};
```

C++ Builder 中实现与 Delphi 中 Variant 兼容的 Variant 类，提供 Variant 纯粹是为了 OLE 自动化。OLE 自动化中使用变体是因为 Microsoft 的 Visual Basic 环境需要这样一种数据类型，这是一种能够允许在任何时刻被赋予任何有效类型数据的类型。可以想象，这种类型是相当慢的，因为它运行时会频繁的进行类型转换，与 C++中其他数据类型相比，Variant 在空间和速度上都耗费很大。

1. 变体数组

实际的 Variant 变量（Variant 类的实例）可以接受多种数据类型，包括一个变体数组。Variant 数组实际上是把一个指针作为它的数据的 Variant 变量，这个指针指向一块为 16 字节 TVarData 结构数组的内存，该数组的每个元素都是一个 Variant，可以装有任何一种类型的数值。显而易见，由于过量的内存占用和速度原因，我们没有必要在通常的编程中使用 Variant 数组，但在给 OLE 自动化对象传递数组数据时，使用 Variant 数组是有必要的。

有两种方式声明一个 Variant 数组，其中一种使用了 VarArrayCreate 函数，如下所示：

```
Variant V(OPENARRAY(int,( 0,9)),varInteger);
Variant V;
V = VarArrayCreate(OPENARRAY(int, (0,3,0,9)), varInteger);
```

在上面的代码中，第一种方法声明了一个一维 Variant 数组，下界为 0，上界 9；第二种方法声明了一个二维 Variant 数组，第一维的上下界分别为 0 和 9，第二维的上下界分别为 0 和 3。

如何使用一个 Variant 数组？还是通过一段代码来说明：

Source 使用变体数组

```
Variant a, b;
int i, tmp;
a = VarArrayCreate(OPENARRAY(int, (0, 9)), varInteger);
b = VarArrayCreate(OPENARRAY(int, (1, 3, 0, 9)), varInteger);
// 通过 ArrayLock 访问数组
int* ax = (int*) a.ArrayLock();
```

```
for (i=0; i < 10; i++)
        ax[i] = i*i;
a.ArrayUnlock();
// 使用 Put/GetElement 访问数组
for (i=0; i < 10; i++)
{
        tmp = i*i*i;
        a.PutElement(&tmp, i);
}
ShowMessage(IntToStr(a.GetElement(5)));
// 2 维 Variant 数组
for (i=0; i < 10; i++)
{
        b.PutElement(&i, 1, i);
        tmp = sqrt(i);
        b.PutElement(&tmp, 2, i);
        tmp = i*i;
        b.PutElement(&tmp, 3, i);
}
ShowMessage(IntToStr(b.GetElement(1,0)));
```

除此之外，C++ Builder 还提供了其他一些 Variant 数组的支持函数，下面是关于这些函数的信息。

- VarArrayRedim 函数。

声明：void __fastcall VarArrayRedim(Variant& A, int high);

改变一变体数组的尺寸。参数 high 为变体数组新的上界，数组中原有元素会被保留下来，新的元素设为 0 或空，Variant 类的 ArrayRedim 方法提供同样功能。

- VarArrayDimCount 函数。

声明：int __fastcall VarArrayDimCount(const Variant &A);

返回一变体数组的维数。Variant 类的 ArrayDimCount 方法提供同样功能。

- VarArrayHighBound/ VarArrayLowBound 函数。

声明：int __fastcall VarArrayHighBound(const Variant &A, int Dim);

返回一变体数组的上界和下界。Variant 类的 ArrayHighBound、ArrayLowBound 方法提供同样功能。

- VarArrayRef 函数。

声明：Variant __fastca0ll VarArrayRef(const Variant &A);

使用 varByRef 标志，返回一个一引用方式与指定变体数组相连的变量。

- VarIsArray 函数。

声明：bool __fastcall VarIsArray(const Variant &A);

判断一个 Variant 变量是否是一个变体数组。

2. 通过 Variant 变量实现自动化

上一节已经接触了使用 Variant 访问自动化服务器的例子，使用 Variant 实现自动化首先要通过 ProgID 创建 OLE 对象，使用 CreateOleObject 函数或者 Variant 类的 CreateObject 方法，例如：

```
OleObj = CreateOleObject ("Project.AutoObject");
```

事实上，CreateOleObject 函数的实现机制并不十分奥妙。它首先调用 Win32 的 CLSIDFromProgId 函数，通过 ProgID 得到自动化对象标识符。而后调用另一个 Win32 函数 CoCreateInstance，传递参数 CLSID，获得一个自动化对象的 IDispatch 接口并返回该接口。

但是，当激活诸如 Microsoft Word 这样的实际应用程序时，可能只是想与当前已运行应用程序连接，而不是重新运行一个副本。为了实现这样的要求，可以使用 GetActiveOleObject 函数，这样做的结果将是仅仅连接已有的应用程序。以下是使用该函数的示例代码：

```
try{
    WordApp = GetActiveOleObject ("Word.Basic");
}
catch(...){
    WordApp = CreateOleObject("Word.Basic");
}
```

如果 GetActiveOleObject 函数运行成功，则会返回对象的 IDispatch 接口的引用。如果 GetActiveOleObject 调用失败，将会抛出一个 EOleSysError 异常。在示例代码中，当出现异常则直接建立新的服务器对象。

一个 OLE 自动化对象具有属性和方法，在前一章中使用了 Variant 类的 Exec 方法访问这些属性和方法。实际上，Variant 类中还包含了 OleFunction、OleProcedure、OlePropertyGet 和 OlePropertySet 方法，提供较为便捷的对象属性和方法的访问方式，例如：

```
WordApp.OleProcedure("ShowApp");
```

实际上，这四个方法是 Exec 方法的封装，它们只支持 10 个以下参数属性和方法的调用。

以下我们创建一个简单的自动化 Word 的例子，这个例子使用了 Word.Basic 对象。Word.Basic 是 Microsoft Word 支持的一个自动化对象。实际上，Word.Basic 是在较低版本的 Word 中推出的，在 Microsoft Word 97 中提供了新的 Word.Application 对象，但 Word.Basic 仍然可用。Word.Basic 使用起来比较简单，但支持的功能比起 Word.Application 对象要少。下面是这个示例程序的关键代码，它将激活计算机中的 Word，并输出一段文字。

Source 使用 Word.Basic 实现简单的 Word 自动化

```
void __fastcall TForm1::TalkToWordClick(TObject *Sender)
{
    Variant WordApp;
    WordApp = CreateOleObject("Word.Basic");
```

```
        WordApp.Exec(Procedure("AppShow"));
        WordApp.Exec(Procedure("FileNew") << "Normal");
        WordApp.Exec(Procedure("Insert") << "Hello from Borland C++Builder 4!");
        WordApp.Exec(Procedure("FileQuit"));
    }
    //---------------------------------------------------------------------------
```

16.5.2 自动化 Excel

Microsoft Office 97（8.0 版）中的应用程序组件包括 Word、Excel、PowerPoint，实际上都是标准的 OLE 自动化服务器。在 Microsoft Office 97 版本中，提供了一组 Application 对象取代了原来的 Basic 对象，Application 对象的功能非常强大，它提供了一系列控制涉及 Office 应用程序的每一个方面。可以说，在实际使用这些 Office 应用程序时所能作的事，无一例外地可以通过 OLE 自动化编写程序完成，图 16-10 是 Excel 动画，图中的三维图表处于不断旋转之中。

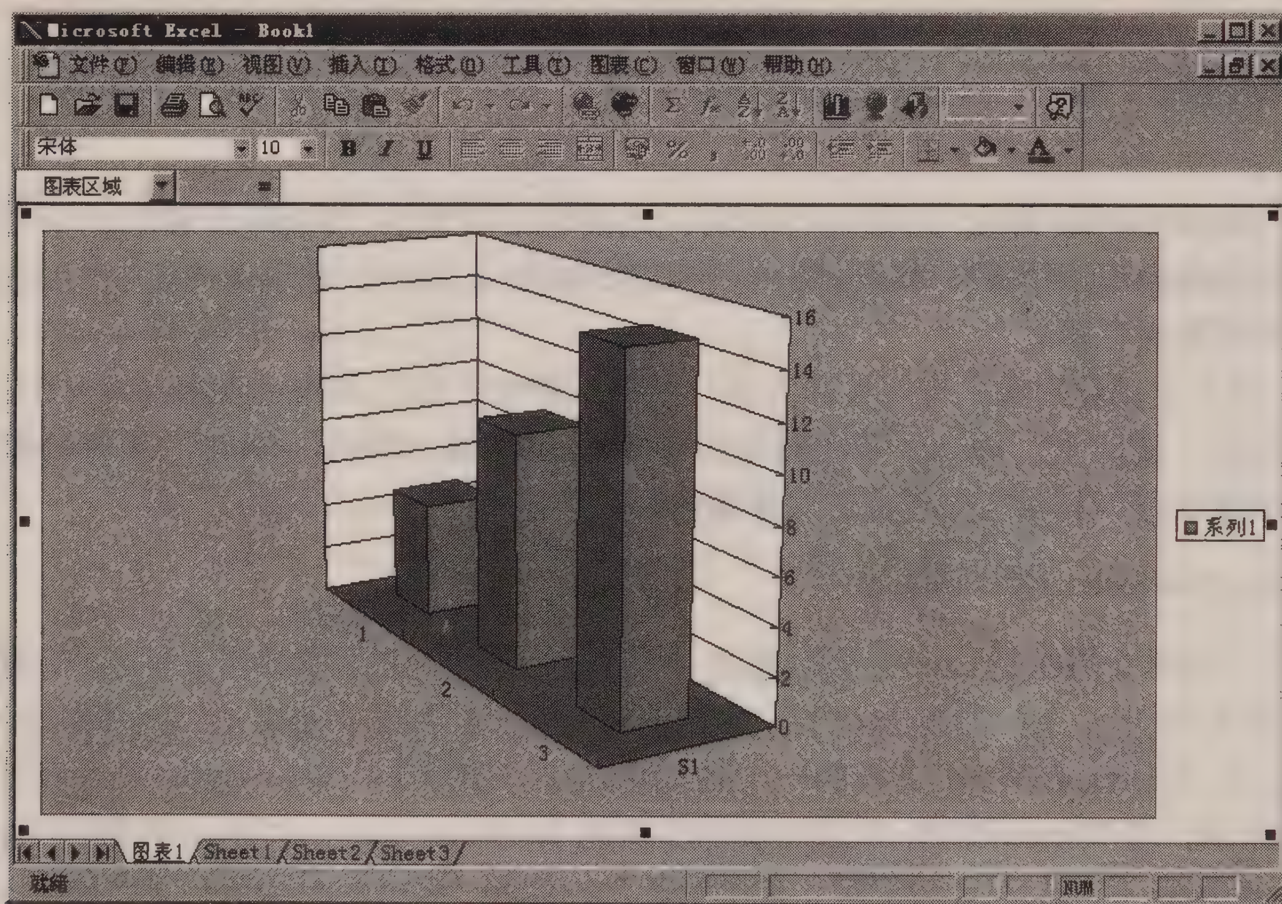


图 16-10 Excel 动画

在此将重点讲述 Excel 自动化服务器的用法。实际上，使用 Excel.Application 对象的许多经验可以推广到 Word.Application 中去。

Excel.Application 的属性很多，比较重要的有 Workbooks、Cells、Range 等等，这些属性与 Excel 应用程序中的数据表格有关。另外，还可以控制另一重要属性 Charts，这个属性用于绘制图表。在下面的例子程序中使用 Excel 完成两项任务：让 Excel 作三维图表旋转动画和向 Excel 发送数据库数据，如图 16-11 所示。

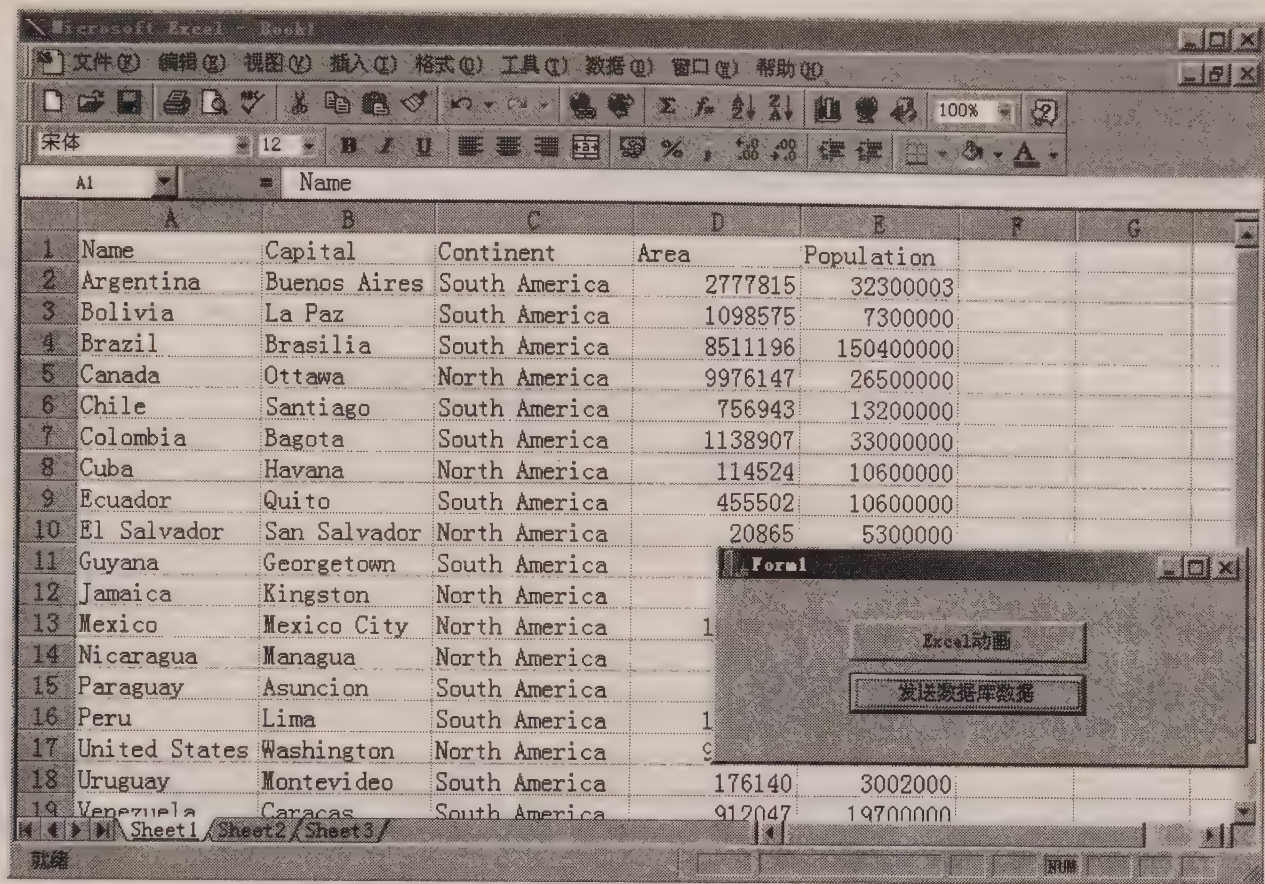


图 16-11　将数据库内容自动在 Excel 中生成电子表格

Source 自动化 Excel

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{       // Excel 动画
        Variant Excel,chart;
        Excel=CreateOleObject("Excel.Application");
        Excel.Exec(PropertyGet("Workbooks")).Exec(Procedure("Add"));
        Excel.Exec(PropertyGet("Cells")<<1<<1).Exec(PropertySet("Value")<<5);
        Excel.Exec(PropertyGet("Cells")<<1<<2).Exec(PropertySet("Value")<<10);
        Excel.Exec(PropertyGet("Cells")<<1<<3).Exec(PropertySet("Value")<<15);
        Excel.Exec(PropertyGet("Range")<<"A1:C1").Exec(Procedure("Select"));
        chart = Excel.Exec(PropertyGet("Charts")).Exec(Function("Add"));
        Excel.Exec(PropertySet("Visible")<<true);
        chart.Exec(PropertySet("Type")<<-4100);
        for(int Rotate=5; Rotate <= 180; Rotate += 5){
                chart.Exec(PropertySet("Rotation")<<Rotate);
        }
        for (int Rotate = 175; Rotate >= 0; Rotate -= 5) {
                chart.Exec(PropertySet("Rotation")<<Rotate);
        }
```

```
        Excel.Exec(Procedure("Quit"));
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{       // 向 Execel 发送数据库表格
        Variant Excel,cell;
        Excel=CreateOleObject("Excel.Application");
        Excel.Exec(PropertyGet("Workbooks")).Exec(Procedure("Add"));
        cell=Excel.Exec(PropertyGet("ActiveCell"));
        Table1->TableName="Country.db";
        Table1->Open();
        Excel.Exec(PropertySet("Visible")<<true);
        for (int i=0;i<Table1->Fields->Count;i++)
        {
                cell.Exec(PropertySet("Value")<<Table1->Fields->Fields[i]->DisplayLabel);
                cell=cell.Exec(Function("Next"));
        }
        Table1->First();
        int row=2;
        while (!Table1->Eof)
        {
                cell=Excel.Exec(PropertyGet("Range")<<
                        ("A"+IntToStr(row)+":A"+IntToStr(row)));
                for (int i=0;i<Table1->Fields->Count;i++)
                {
                        cell.Exec(PropertySet("Value")<<Table1->Fields->Fields[i]->AsString);
                        cell=cell.Exec(Function("Next"));
                }
                Table1->Next();
                row++;
        }
        Excel.Exec(Procedure("Quit"));
        Table1->Close();
}
//---------------------------------------------------------------------------
```

这段程序的效果可以从图 16-10 和图 16-11 中看到。第一段代码在 Excel 中建立了一个三维图表，并通过图表对象的 Rotation 属性让这个三维图表旋转两周，形成动画效果。让 Excel 作动画充分显示了 OLE 自动化的威力，这种效果显然是使用 Excel 时无法做到的。第二段代码具有现实意义，它自动处理一个数据库，并将数据库的内容在 Excel 中生成电子表格显示出来。

16.5.3 内部自动化应用程序

在 16.4 节中我们讨论了三种访问自动化对象的方法：使用 Variant（IDispatch）、dispinterface 和接口。前文所示的自动化 Word 和 Excel 的实例程序都是采用 Variant 方法进行访问的，但我们知道，使用 Variant 变量访问自动化对象的效率是最低的。是否可以使用 dispinterface 或者智能接口操纵实际的 OLE 服务器（如 Word、Excel）？答案是肯定的，因为 C++ Builder 可以引入 Word 或 Excel 的类型库，并自动分析该类型库生成类型库头文件和 CPP 文件。

当使用 Office 2000 时，可以引入 Office 2000 的几个应用程序的类型库，这是带有 OLB 扩展名的文件，用于 Word 的文件是 MsWord9.olb，用于 Excel 的文件是 Excel9.olb，此外还可以引入 PowerPoint 的类型库。引入类型库工作将会生成可以让 C++ Builder 使用接口访问 Word 或 Excel 服务器的头文件（参见图 16-12）。

在 Borland C++ Builder 5 中，我们可以更为方便地调用自动化对象，即 C++Builder 提供对 COM 对象进行 VCL 组件形式的封装，并使自动化对象乃至 COM 对象的访问更为简单。C++Builder 支持在引入类型库时将其中包含的 COM 对象封装成以 TOleServer 类为父类的组件。TOleServer 类提供的 Connect 和 Disconnect 方法，支持简单有效的创建对象实例或销毁对象实例。实际上，TOleServer 类可以认为是 CoClass 客户代理类的封装，它以明晰的方式实现了 CoClass 的默认方法和事件。

对于 Office 对象、Shell 对象这样的常用 COM 对象，在安装 BCB 时即被封装称为 VCL 组件的形式，这些组件在 C++Builder 组件选项板的 Server 页。

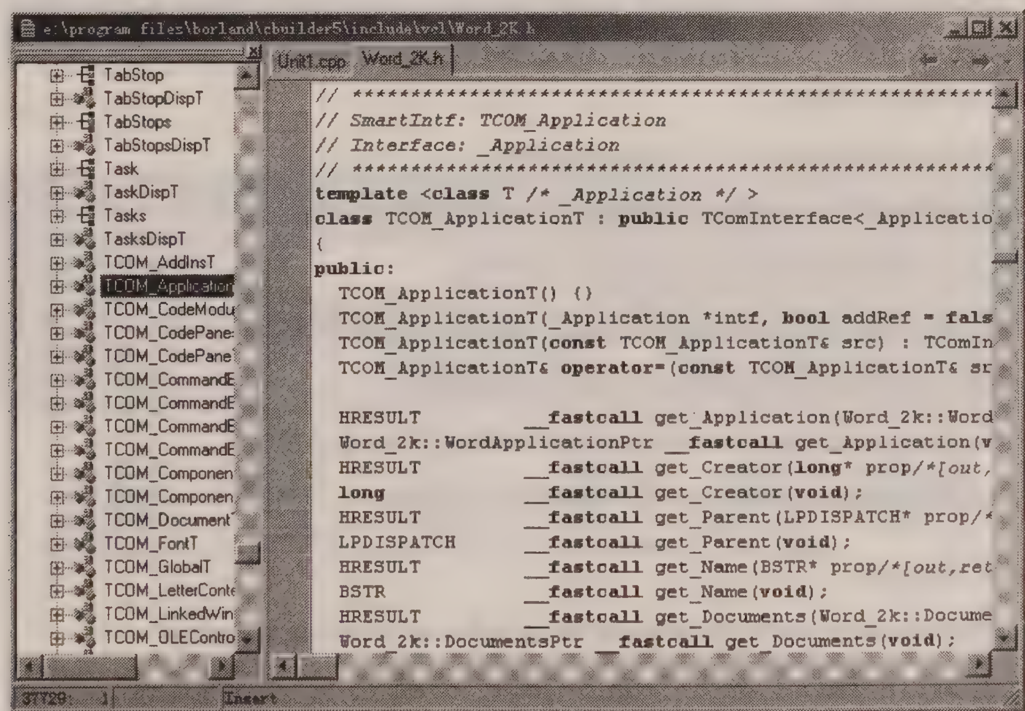


图 16-12 由 Word 2000 类型库生成的 Word_2K.h 头文件（包括了所有 Word 对象信息）

在此，将创建使用 Word 艺术字功能的范例程序。该程序通过 BCB 封装的 Office 组件，利用效率最高的智能接口方式控制 Word 自动化服务器。

Source 使用接口进行 Word 的内部自动化

```
//---------------------------------------------------------------------------
```

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
        WordApp=CoApplication_::Create();
        WordApp->Documents_->Add()->Select();
        WordApp->ActiveDocument->Shapes_->AddTextEffect
              (0,Variant("高级实战开发"),Variant("隶书"),48,
              -1,0,Image1->Width,Image1->Height)->Select();
        ComboBox1->Items = Screen->Fonts;
        ComboBox1->Text="隶书";
        ShowFont();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
        WordApplication1->Quit(Variant(false));
        WordApplication1->Disconnect
}
//---------------------------------------------------------------------------
void __fastcall TForm1::ShowFont()
{
        AnsiString t = Edit1->Text;
        t = t +AnsiString::StringOfChar(' ',12-t.Length());
WordDocument1->Range(EmptyParam,EmptyParam)->ShapeRange
        ->TextEffect->Text=Variant(t);
WordDocument1->Range(EmptyParam,EmptyParam)->ShapeRange
        ->TextEffect->PresetTextEffect=UpDown1->Position;
WordDocument1->Range(EmptyParam,EmptyParam)->ShapeRange
        ->TextEffect->FontName=Variant(ComboBox1)->Text);
if(CheckBox1)->Checked)
        WordDocument1->Range(EmptyParam,EmptyParam)->ShapeRange
        ->TextEffect->FontBold=-1;
else
        WordDocument1->Range(EmptyParam,EmptyParam)->ShapeRange
        ->TextEffect->FontBold=0;
if(CheckBox2->Checked)
        WordDocument1->Range(EmptyParam,EmptyParam)->ShapeRange
        ->TextEffect->FontItalic=-1;
```

```
else
    WordDocument1->Range(EmptyParam,EmptyParam)->ShapeRange
    ->TextEffect->FontItalic=0;
WordDocument1->Range(EmptyParam,EmptyParam)->Copy AsPicture();
TMetafile *Metapic=new TMetafile();
Metapic->Assign(Clipboard());
    TMetafile *Metapic=new TMetafile();
    Metapic->Assign(Clipboard());
    Image1->Picture->Assign(Metapic);
    delete Metapic;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    ShowFont();
}
//---------------------------------------------------------------------------
```

使用 Word 智能接口的步骤和通常的 COM 对象是相同的。首先声明接口类，为 TCOM_Application 类型，该类型等价于“Word.Application”，然后通过 CoClass 客户代理类创建一个自动化对象实例，余下的工作就是使用接口访问 Word 服务器了，通过类型库头文件可以得到关于如何使用自动化对象足够信息。图 16-13 使用接口进行 Word 的内容自动化，程序展示如何使用 Word 的艺术字功能。

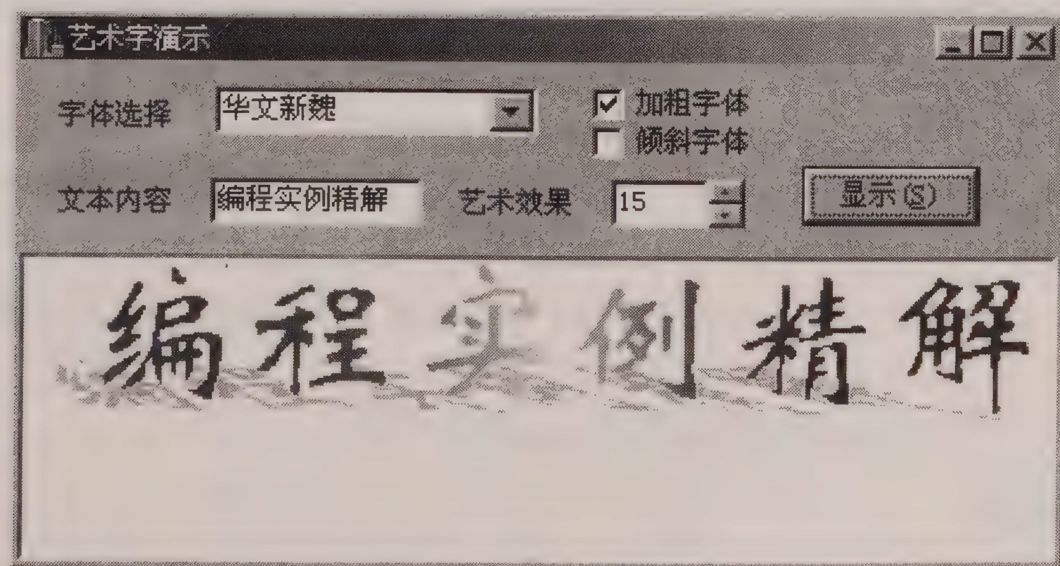


图 16-13　使用接口进行 Word 的内部自动化

这段程序并没有设置 WordApp 的 Visible 属性，所以 Word 服务器完全是在后台运行的。程序主要通过 TextEffect 属性实现艺术字效果，Word 提供 30 种预制艺术字类型，通过 PresetTextEffect 属性可以进行设置。但如何将 Word 中的艺术字在 C++应用程序中显示出来呢？这里用了一点技巧：在 Word 选中设置好的艺术字（Select 方法），然后使用 CopyAsPicture 将艺术字复制到剪贴板，接下来从剪贴板中将图像取出，在本节创建的应用程序中显示出来。因为艺术字图像为图元文件格式，所以该程序中必须创建一个临时 TMetaFile

对象，才能使用 Assign 方法获得艺术字图案。

另外需要注意的是，由于为方便 Visual Basic 访问，一些布尔属性以 0 表示真，以-1 表示假。例如本例中的 FontBold 和 FontItalic，在程序中必须处理这些属性的设置。

使用 dispinterface 访问 Word 或 Excel 服务器也不困难的，下面的示例代码实现了通过 dispinterface 创建自动化对象实例：

```
_ApplicationDisp WordApp;
WordApp.BindDefault();
```

16.6 Internet Explorer 控件的高级用法

在第 10 章中，我们利用了 Microsoft Internet Control 这个 ActiveX 控件创建了 BCB Web Browser 程序的第二版本。Internet Control 控件是随 Internet Explorer 4.0 得到的，它提供了与 Internet Explorer 4.0 相同的强大功能，所以利用这个 ActiveX 控件创建的浏览器的浏览效果极佳。但在第 10 章中，由于某些知识上的缺乏，没有能够完成一些必要的辅助功能，如查看源文件等。其实，使用 Internet Control 控件是完全能够做到这一点的，并且它可以实现更加强劲的控制功能，直到 Web 页面上的每一个元素。这些功能都源于该控件的一个重要属性 Document，而这个属性实际上是一个 COM 接口，通过引入相应的类型库，就可以访问这个重要属性。

前文中曾多次遇到了类型库这个概念，当时并没有对这个概念进行讲述。本节首先将讨论类型库的有关问题，以及 C++ Builder 类型库编辑器的使用。然后将创建 BCB Web Browser 程序的第三版本，加入查看 HTML 信息和源文件功能。

16.6.1 类型库（Type Library）

在 C++ Builder 中创建 COM 对象、OLE 自动化对象、ActiveX 控件都要涉及类型库（Type Library）的建立和编辑，类型库实际上是一个文件，它包含 COM 对象中的数据类型、接口、成员函数等信息，使用 C++ Builder 提供的向导创建标准 COM 对象、OLE 自动化对象、以及 ActiveX 控件时，这些向导会自动生成类型库文件（TLB），并作为资源连接到所要创建的 DLL 或 EXE 中，让其他应用程序或其他编程工具识别和使用。

1. 类型库编辑器

类型库的主要内容是关于 COM 对象的数据类型、接口和成员函数的信息，包括 COM 对象的 IID、CLSID、DispID。类型库中还包括自定义接口的类型信息、由 ActiveX 服务器输出的例程以及对其他类型库的引用。

C++ Builder 包含了一个出色的工具——类型库编辑器（Type Library Editor）为观察和编辑类型库提供可视化支持，类型库编辑器的出现使我们不必再去学习脚本语言 IDL 或 ODL 建立类型库。类型库编辑器提供友好的界面，能够交互生成和编辑类型库信息，并可以自动生成对应的 C++源代码（在建立 COM 工程时）。

编辑当前工程的类型库选择【View/Type Library】。要观察和编辑一个非当前项目的类型库，例如，想通过类型库编辑器查看 Word 或 Excel 的类型库信息，应选择【File/Open】，再把文件类型变为 Type Library 列出能够装有类型库信息的文件，打开文件，C++Builder 将自动激活类型库编辑器并装入所选的类型库。

2. 类型库信息

类型库编辑器左侧树视图的最顶层节点表示类型库自身。选择该节点，在右边的窗格将显示类型库的一般信息，全部信息分为“Attributes”、“Uses”、“Flags”和“Text”四种显示。

Attributes 页显示类型库的属性信息。其中 Name 为类型库名称，GUID 为类型库的全局唯一标识符，Version 为类型库版本号，由主版本号和次版本号两部分组成。LCID 表示类型库中字符串的所用语言。另外，还包含 Help String、Help File、Help Context、Help String DLL、Help String Context 几项为该类型库帮助信息。

Uses 页显示该类型库还引用了其他哪些类型库。例如 Word8 的类型库还引用了 Office8 对象类型库等 4 个其他类型库。

Flags 页用于查看和设置类型库标志。以下是可能的标志及其含义：

Restricted	表示该类型库不能用于宏编程环境，如 Visual Basic。
Control	指出该库代表一个控件。
Hidden	表示类型信息虽然存在，但却是隐藏的，不能显示出来。

Text 页用于显示和编辑类型库中某个对象的 IDL 脚本，IDL 脚本修改的结果会反映到类型库编辑器中。

3. 接口和 DispInterface

类型库编辑器的第一和第二个按钮分别用于添加接口和添加 dispinterface。对于接口和 dispinterface，右侧的窗格会显示“Attributes”、“Flags”和“Text”三页信息。Attributes 页和 Text 页的含义与类型库元素基本相同。在 Flags 页中，可以设置关于接口和 DispInterface 的标志信息，以下是可能的标志及其含义：

Hidden	表示接口存在，但不显示出来。
NoneExtensible	指出 IDispatch 只实现接口描述中列出的属性和方法。
Dual	指出该接口为双重接口，同时支持 IDispatch 接口和虚拟函数表。
OLE Automation	表示接口只能使用与 OLE 自动化相容的数据类型。

我们可以为接口加入成员，包括接口方法和属性。使用工具栏上的一组按钮可以方便的实现这一点。具体创建和编辑接口成员的方法在前文中已经作过讲述。

4. 组件类（CoClass）

类型库组件类（Component Class 或 CoClass）代表整个自动化对象、定制 COM 对象或 ActiveX 控件。它装有提供给用户的接口和 dispinterface。可以单击类型库编辑器工具栏的 New

CoClass 按钮建立一个组件类。

对于组件类，在编辑器右边窗格中也包含 Flags 页，用于设置组件类的标志，以下是可能的标志及其含义：

CanCreate	支持该组件类可由 CoCreateInstance 创建实例。
ApplicationObject	指出组件类只用于 Out-of-Process 类型的自动化服务器。
Licensed	该标志用于 ActiveX 控件，表示在设计期或运行期都需要许可，组件类的实例只能由 IClassFactory2 创建。
Predefined	指出客户程序应当自动创建 OLE 对象的单个实例。
Control	表示此组件类为一控件。
Aggregatable	指出组件类的成员可以被聚合。
Replaceable	指出此组件类支持 IConnectionPointWithDefault。

16.6.2 BCB WebBrowser 的第 3 版本

在第 10 章中，曾经使用 Microsoft Internet Control 控件创建了 BCB Web Browser 的第二版本，这个 ActiveX 控件包含了 Document 属性，但当时还不能使用它。实际上，Document 属性为一个 IHTMLDocument2 类型的接口，这个接口的定义必须通过引入类型库得到。

Document 属性包括了对 Web 页面元素的全面控制，包括了使用 JavaScript 和 VBScript 可以访问或不能访问的全部内容，并包含所有 DHTML 属性的控制（如 InnerHTML）。现在，我们开始着手创建 BCB WebBrowser 程序的第 3 版，通过 IHTMLDocument2 接口实现以前不曾实现的功能。图 16-14 显示了程序通过接口实现查看源文件的功能。

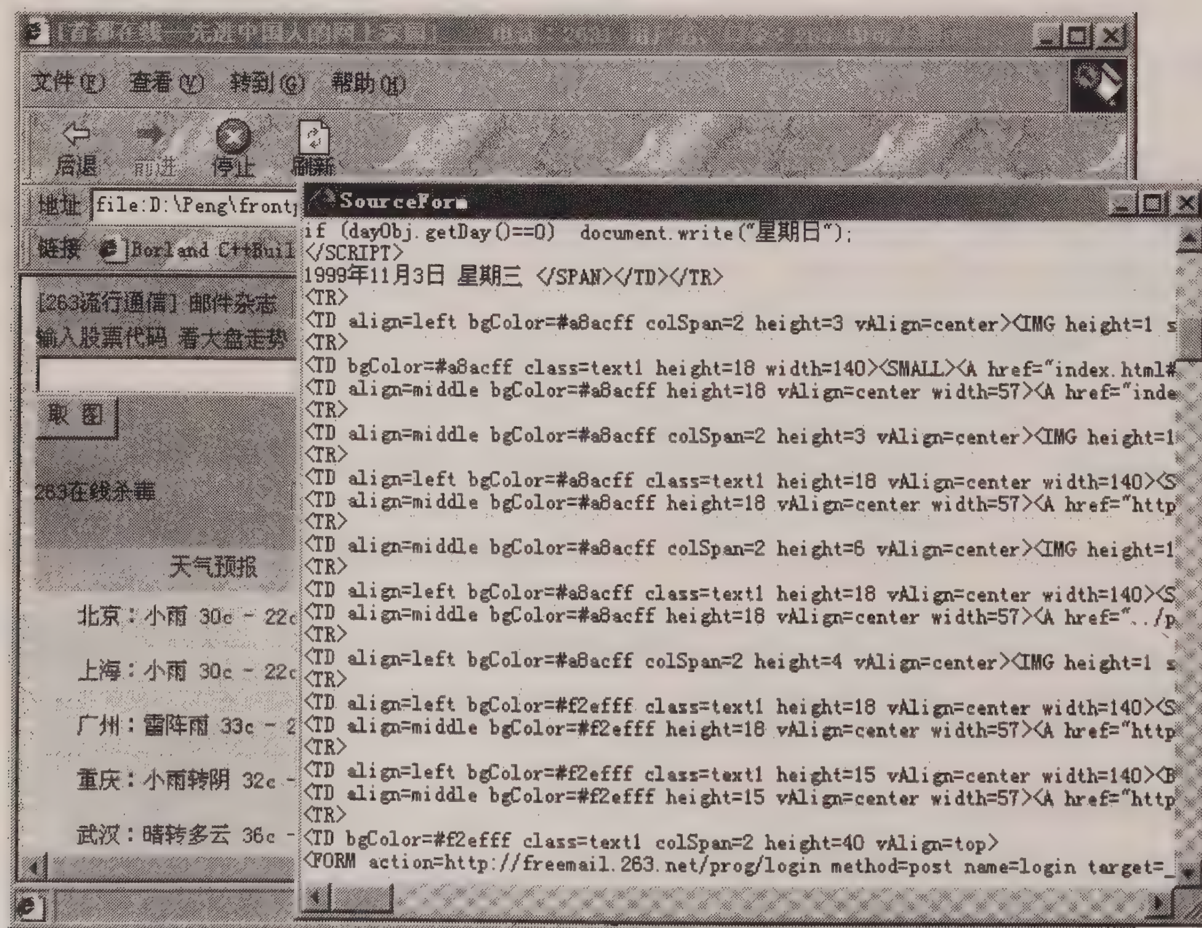


图 16-14 通过接口实现查看源文件功能

首先，在 BCB WebBrowser2 工程中使用【**Import Type Library**】引入 Microsoft HTML Object Library (Version 4.0)类型库，该库包含了 IHTMLDocument2 接口的定义。包含类型库头文件 MSHTML_TLB.h，并为主窗口类加入成员变量 Doc 作为 Document 属性的引用：

```
IHTMLDocument2* Doc;
```

这里基于 BCB WebBrowser2 增加了两项新的功能，查看 HTML 详细信息和查看源文件功能。增加的主要代码如下。

Source 通过接口实现 WebBrowser 控件的高级使用

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::T1Click(TObject *Sender)
{
        Doc=(IHTMLDocument2*)(IUnknown*)WebBrowser1->Document;
        SourceForm->Memo1->Text=Doc->body->outerHTML;
        SourceForm->Show();
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::ShowDetails()
{
        Doc=(IHTMLDocument2*)(IUnknown*)WebBrowser1->Document;
        DetailsForm->Edit1->Text=Doc->get_bgColor();
        DetailsForm->Edit2->Text=Doc->get_fgColor();
        DetailsForm->Edit3->Text=Doc->get_linkColor();
        DetailsForm->Edit4->Text=Doc->get_vlinkColor();
        DetailsForm->Edit5->Text=Doc->fileUpdatedDate;
        DetailsForm->Edit6->Text=Doc->fileSize;
        DetailsForm->Edit7->Text=Doc->fileCreatedDate;
        DetailsForm->Edit8->Text=Doc->fileModifiedDate;
        DetailsForm->Edit9->Text=Doc->title;
        DetailsForm->Edit10->Text=Doc->url;
        DetailsForm->Edit11->Text=Doc->domain;
        DetailsForm->Edit12->Text=Doc->protocol;
        DetailsForm->Edit13->Text=Doc->cookie;
        DetailsForm->Edit14->Text=Doc->security;
        DetailsForm->Edit15->Text=Doc->charset;
        DetailsForm->Edit16->Text=Doc->defaultCharset;
        DetailsForm->Edit17->Text=Doc->mimeType;
        DetailsForm->Edit18->Text=Doc->nameProp;
}
//---------------------------------------------------------------------------
```

```
void __fastcall TMainForm::N7Click(TObject *Sender)
{
    ShowDetails();
    DetailsForm->Show();
}
//--------------------------------------------------------------------------------
```

为使用 IHTMLDocument2 接口，首先必须对 Document 属性进行转换，Document 属性声明为_di_IDispatch，转换不能直接进行。但可以先将该属性转换为 IUnknown，进而转换为 IHTMLDocument2 类型。IHTMLDocument2 页是一个相当复杂的接口，具体内容请参看类型库头文件。

程序中首先实现了查看源文件功能，这是不难的，通过访问 boby 属性的 outerHTML 即可，在此创建新的窗体 SourceForm 以显示 HTML 源文件。

其次，我们实现了查看 HTML 信息功能，为此建立 DetailsForm 窗体，该窗体放置了一组 Label 和编辑框。显示这些信息是通过直接访问 Document 的一些属性完成的（如图 16-15 所示）。

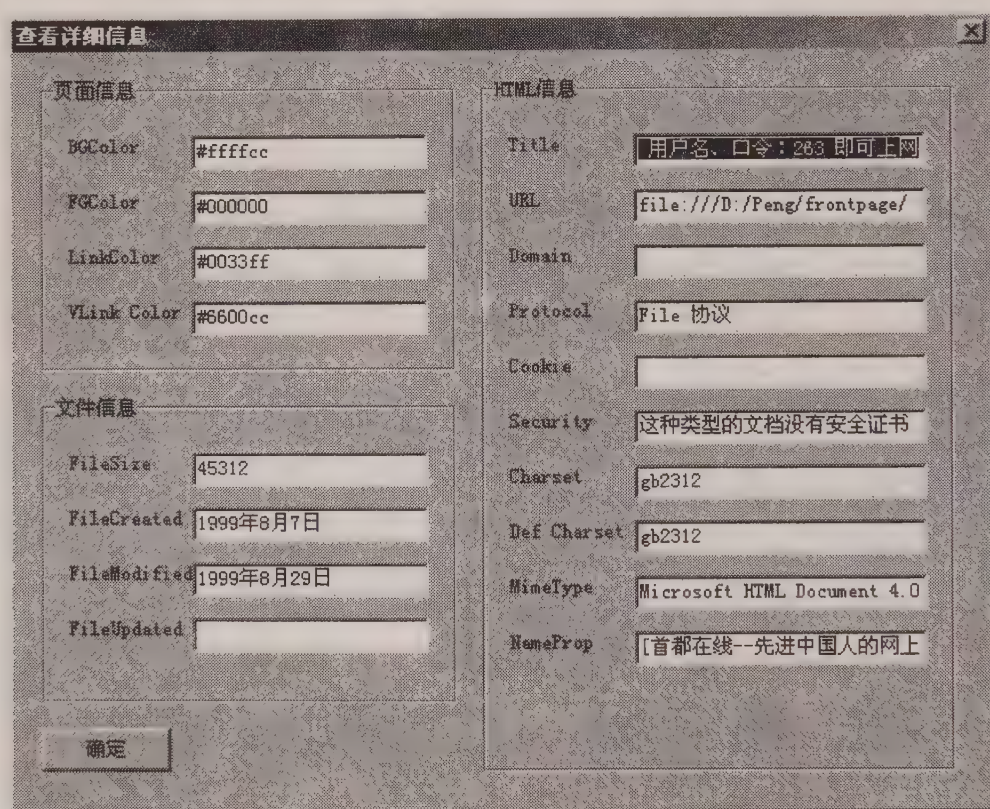


图 16-15　查看 HTML 详细信息功能

在这段程序中访问 IHTMLDocument2 属性是比较简单的。实际上，页面元素属性（如 all）包含了 item 属性，通过这个属性可以遍历某一元素的所有子元素。总之，通过 IHTMLDocument2 接口完全可以控制 Web 页面的每一个方面。

16.7　ActiveX 技术和创建 ActiveX 控件

C++ Builder 使用组件编程，并且可以方便的创建新的 VCL 组件。但是，C++ Builder 同样也支持 ActiveX 控件，在前面的章节我们已经看到了如何在 C++ Builder 程序中使用 ActiveX 控

件。本节将更为深入的讨论 ActiveX 技术和 ActiveX 控件的创建问题。

虽然 C++ Builder 支持 ActiveX 控件，但在 C++ Builder 编程中，更多的是使用 VCL 组件创建程序。VCL 组件一向工作的很好，那么，VCL 组件和 ActiceX 控件有什么技术区别呢？

本质上，ActiveX 基于 COM 技术，可以被认为是一种特殊的 In-Process 服务器，这种服务器存在于以 OCX 为扩展名的动态链接库中。ActiveX 控件基于 DLL，这就意味着 ActiveX 控件只能与应用程序进行动态链接，而 C++ Builder 中的 VCL 组件则可以选择静态或动态链接。

相对于 VCL 组件来说，ActiveX 控件非常臃肿，而且效率并不高（ActiveX 组件必然是一个窗口，而 VCL 组件则可以是非窗口组件，非窗口组件的资源占用要低得多）。同时 VCL 组件还有一些 ActiveX 控件无法相比的优越特性。但是，VCL 组件的致命弱点是，它仅仅能够在 C++ Builder 中使用，而 ActiveX 控件却被广泛地支持。ActiveX 控件可以在各种开发环境中使用，甚至还可以嵌入 Web 页中，这些特点是 VCL 组件无法比拟的。

幸运的是，C++ Builder 中创建 ActiveX 控件毫不费力。C++ Builder 创建 ActiveX 控件甚至比 Visual Basic 还要容易。这表现在两种非常易用的 ActiveX 控件创建方式：

(1) 可以使用 ActiveX Control Wizard 将 VCL 组件自动转化为一个 ActiveX 控件。在这个过程中，C++ Builder 完成了大量的工作。由于这种创建方式的存在，我们可以先从创建一个 VCL 组件开始，然后将其转化为功能相同的 ActiveX 控件。唯一的条件是这个 VCL 组件必须是 TWinControl 的某一级派生类。

(2) 可以通过建立一个 ActiveForm，在其中放置一些组件，并携带整个窗体作为一个 ActiveX 控件。这实际上是 Visual Basic 中创建 ActiveX 控件的方式。这也是一个简单有效的创建方法。

16.7.1 创建 TCoolClock 的 ActiveX 版本

在本书第 13 章中完成的 TCoolClock 组件继承于 TCustomControl 类，所以它满足自动转化为 ActiveX 组件的条件。现在就着手创建 ActiveX 控件，这一切并不需要做太多的工作。

在确认 TCoolClock 组件正常安装后，首先选择菜单【**File/New**】命令，在 ActiveX 页面上选择 ActiveX Library 创建一个 ActiveX 工程，并将其保存为 XCoolClock。

然后可以在 ActiveX 页面上选择 ActiveX Control，在弹出的 ActiveX Control Wizard 对话框中选择转换的 VCL 类。填写完毕后单击 OK 按钮，C++ Builder 就会自动转换 ActiveX 控件的完整源代码。

ActiveX Control Wizard 对话框下方存在三个复选框，如图 16-16 所示。如果选中 Include Design-time License 框，ActiveX Control Wizard 会自动产生控件许可证号码，并将其存入.LIC 文件中；如果选中 Include Version Information 框，ActiveX 控件将包含版本信息；如果选中最后一个复选框，ActiveX Control Wizard 会自动为控件增加一个 About 对话框。

ActiveX Control Wizard 生成的实现单元文件是 CoolClockImpl.cpp，该单元中包含了重写的 ActiveX 控件源代码，下面是源代码的一部分。

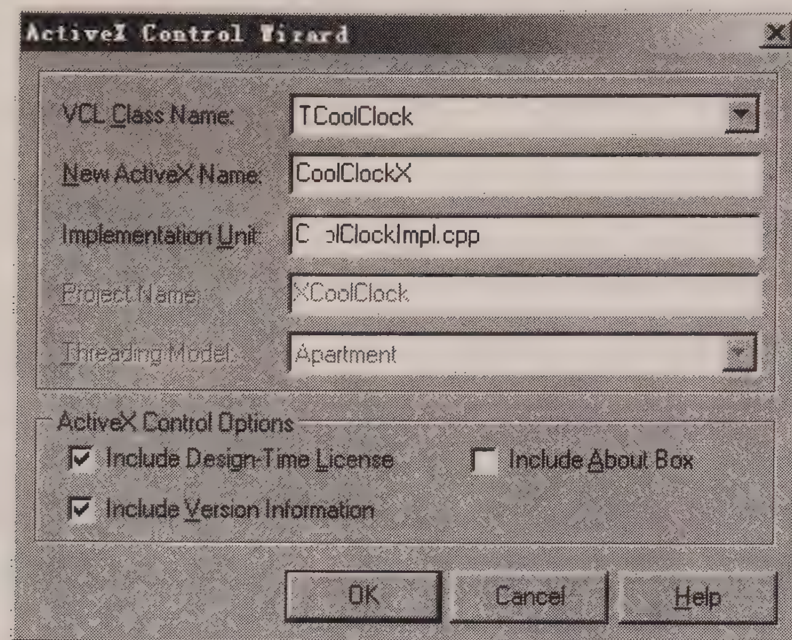


图 16-16　ActiveX Control Wizard 对话框

```
// 重新定义的读方法
STDMETHODIMP TCoolClockXImpl::get_Active(TOLEBOOL* Value)
{
    try
    {
        *Value = m_VclCtl->Active;
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_ICoolClockX);
    }
    return S_OK;
};
STDMETHODIMP TCoolClockXImpl::get_ClockStyle(TxClockStyle* Value)
{
    try
    {
        *Value = (TxClockStyle)(m_VclCtl->ClockStyle);
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_ICoolClockX);
    }
    return S_OK;
};
…………
```

```
// 重新定义的写方法
STDMETHODIMP TCoolClockXImpl::set_Active(TOLEBOOL Value)
{
    try
    {
        const DISPID dispid = 3;
        if (FireOnRequestEdit(dispid) == S_FALSE)
            return S_FALSE;
        m_VclCtl->Active = Value;
        FireOnChanged(dispid);
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_ICoolClockX);
    }
    return S_OK;
};
STDMETHODIMP TCoolClockXImpl::set_ClockStyle(TxClockStyle Value)
{
    try
    {
        const DISPID dispid = 6;
        if (FireOnRequestEdit(dispid) == S_FALSE)
            return S_FALSE;
        m_VclCtl->ClockStyle = (TClockStyle)(Value);
        FireOnChanged(dispid);
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_ICoolClockX);
    }
    return S_OK;
};
............
// 重新定义的事件
void __fastcall TCoolClockXImpl::KeyPressEvent(TObject *Sender, char &Key)
{
    signed_char TempKey;
    TempKey = (signed_char)Key;
    Fire_OnKeyPress(&TempKey);
```

```
        Key = (signed_char)TempKey;
    };
```

此时可以对 ActiveX 控件源代码进行编译了。但为了能够使 ActiveX 控件编译通过，我们不得不放弃 TCoolClock 的一点功能，删除原来 TCoolClock 组件中关于 OnMouseEnter 和 OnMouseLeave 事件的实现部分，C++ Builder 的内部事件无法在 ActiveX 控件中得到支持。

编译成功后，应该选择菜单【**Run/Register ActiveX Server**】注册 ActiveX 控件，注册完毕后，ActiveX 控件 CoolClockX 就可以安装在其他开发环境中了。图 16-17 显示了在 Visual C++ 中使用 CoolClockX 控件的情景。

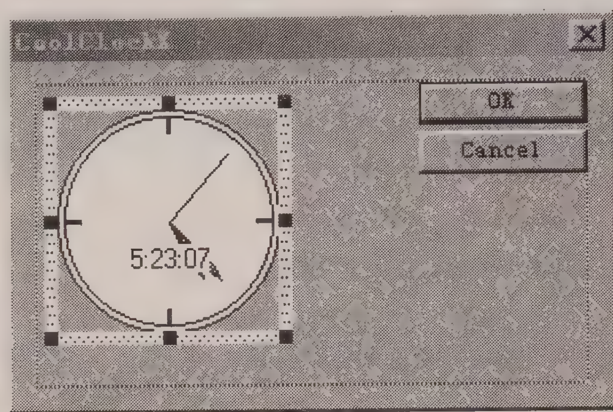


图 16-17　在 Visual C++中使用 CoolClockX 控件

16.7.2　为 ActiveX 控件添加属性

ActiveX Control Wizard 做的关键工作是创建一个类型库，打开 C++Builder 的类型库编辑器（菜单项【**View/Type Libray**】），可以看到 ActiveX 的类型库信息。

通过类型库编辑器，可以对 ActiveX 控件工程进行管理。包括为 ActiveX 控件方便的加入属性或方法。

将 VCL 组件转换为 ActiveX 控件后，许多原来可用的属性都失去了，产生这种情况是由于两个原因：

(1) ActiveX 控件对 VCL 组件中的许多特定类型无法支持，例如本例中的 TPen，Tbrush 等。

(2) ActiveX 控件的属性读取必须使用方法，而不能像 VCL 组件中直接读取内部数据域的值。

作为添加属性的演示，下面将为 CoolClockX 控件增加一个 Color 属性，它对应着原组件中 Brush 属性的 Color 属性。这是有必要的，因为在 CoolClock 的 ActiveX 控件版本中 Brush 属性已不可用。

首先在类型库编辑器中单击上方的 New Property 图标，C++ Builder 将自动生成缺省名为 Property1 的属性的读方法和写方法。将属性名改为 Color，然后在左边的 Parameters 选项页中设置方法的参数和返回值类型，在 Flag 页中设置属性的标志，也可以在 Attributes 页设置属性名和属性 ID，以及与帮助文件关联的信息。

对于 Color 属性来说，因为 VCL 中的 TColor 类型已不可用，所以设置参数类型为 long

（四字节整型）。图 16-18 展示了通过类型库编辑器 Type Library 为控件添加属性。

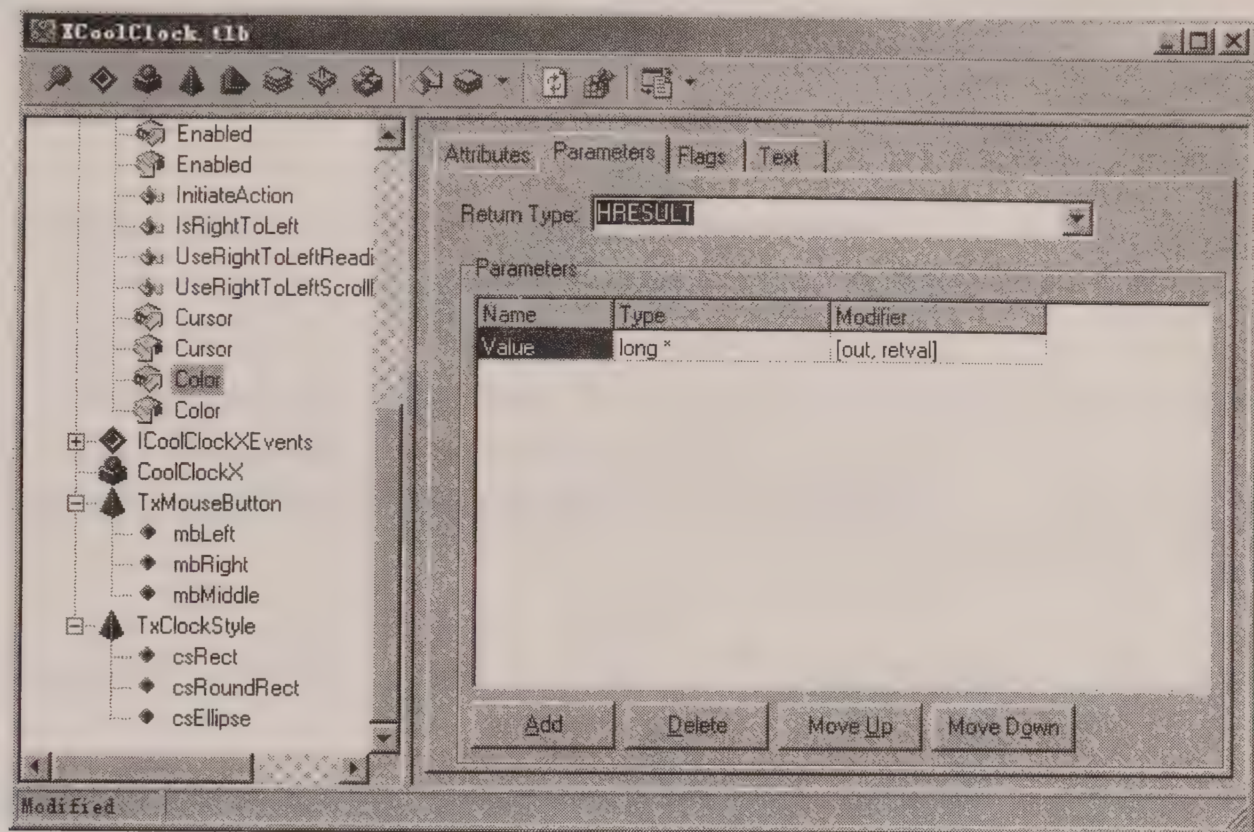


图 16-18　通过类型库编辑器 Type Library 为控件添加属性

在对属性的描述设置完成后，单击 Refresh Implementation 图标刷新，C++ Builder 将自动生成属性读写方法框架。然后切换到主单元文件，在 Color 属性的读写方法中加入功能实现代码。Color 属性完成后的 Set 和 Get 对象方法如下：

```
STDMETHODIMP TCoolClockXImpl::get_Color(long* Value)
{
    try
    {

        *Value = ColorToRGB(m_VclCtl->Brush->Color);
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_ICoolClockX);
    }
    return S_OK;
};
STDMETHODIMP TCoolClockXImpl::set_Color(long Value)
{
    try
    {
        const DISPID dispid = 6;  // 属性 DispID
        if (FireOnRequestEdit(dispid) == S_FALSE)
```

```
            return S_FALSE;
        m_VclCtl->Brush->Color = (TColor)(Value);
        FireOnChanged(dispid);
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_ICoolClockX);
    }
    return S_OK;
};
```

现在重新编译该控件，可以看到 CoolClockX 控件已具有 Color 属性，并且可以在程序中使用 RGB 宏建立新的颜色值。

16.7.3 为 ActiveX 控件编写属性页

ActiveX 控件可以包含属性页，所谓属性页是用于修改 ActiveX 控件属性的对话框。对于各种编程环境，包括 Visual Basic、Visual C++以及 C++ Builder 中，在 ActiveX 控件上单击鼠标右键，在弹出的菜单中选择“Properties”命令，就会打开该 ActiveX 控件的属性页。

下面还是以 XCoolClock 控件为例，介绍创建属性页的一般步骤：单击【**File/New**】菜单项，在 New Items 对话框的 ActiveX 页中双击 Property Page 图标，向导将创建一个空白的属性页 PropertyPage1、一个对应的单元文件 Unit1。

新创建的属性页窗体是空白的，可以根据要编辑的属性把合适的 VCL 控件加到 Form 上，例如，对于字符串类型的特性，一般用 TEdit 组件来编辑，对于枚举值属性，使用 TComboBox 组合框为用户提供选择。在一些情况下，可能需要用多个 VCL 控件配合起来编辑。在本例中，设计的属性页窗体如图 16-19 所示。

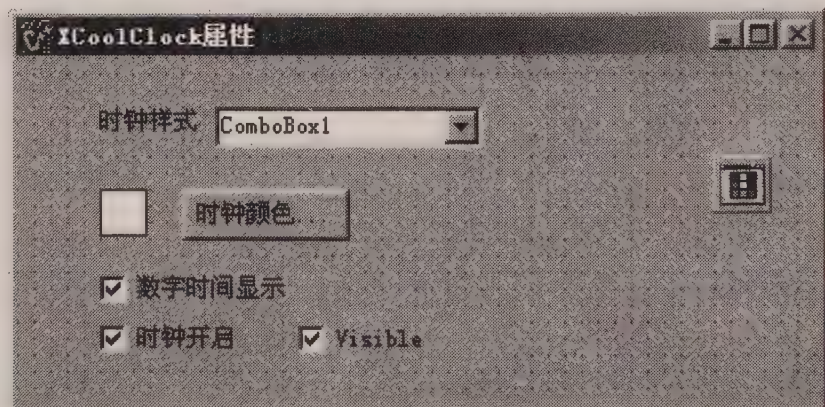


图 16-19　可视化设计属性页窗体

在属性页窗体设计好后，必须把 ActiveX 控件的属性与属性页上的 VCL 组件联系起来，这就要在属性页单元中重载 TPropertyPage 的 UpdatePropertyPage 和 UpdateObject，其中，UpdatePropertyPage 用于从 ActiveX 控件中读属性值到属性页上，而 UpdateObject 用于把属性页上的值写到 ActiveX 控件中。

以下是实现属性页操纵 ActiveX 控件属性的代码。

Source 属性页操纵 ActiveX 控件属性的实现代码

```
void __fastcall TPropertyPage1::UpdatePropertyPage(void)
{
	CheckBox1->Checked=OleObject.Exec(PropertyGet("Active"));
	CheckBox2->Checked=OleObject.Exec(PropertyGet("ShowTime"));
	CheckBox3->Checked=OleObject.Exec(PropertyGet("Visible"));
	ComboBox1->ItemIndex=(int)OleObject.Exec(PropertyGet("ClockStyle"));
	long color=OleObject.Exec(PropertyGet("Color"));
	Shape1->Brush->Color=TColor(color);
}
//---------------------------------------------------------------------------
void __fastcall TPropertyPage1::UpdateObject(void)
{
	OleObject.Exec(PropertySet("Active")<<CheckBox1->Checked);
	OleObject.Exec(PropertySet("ShowTime")<<CheckBox2->Checked);
	OleObject.Exec(PropertySet("Visible")<<CheckBox3->Checked);
	OleObject.Exec(PropertySet("ClockStyle")<<ComboBox1->ItemIndex);
	OleObject.Exec(PropertySet("Color")<<Shape1->Brush->Color);
}
//---------------------------------------------------------------------------
void __fastcall TPropertyPage1::Button1Click(TObject *Sender)
{
	ColorDialog1->Color=Shape1->Brush->Color;
	if (ColorDialog1->Execute()){
		Shape1->Brush->Color=ColorDialog1->Color;
	}
	Modified();
}
//---------------------------------------------------------------------------
```

对于 Color 属性，程序中使用 TShape 组件和 TColorDialog 对话框提供这个属性的可视化编辑。比较特别的是，改变窗体中 TShape 组件的值并不能直接导致属性页 Modified 方法的调用（而对于属性页窗体的 CheckBox 和 ComboBox，在组件属性被改变时，将自动通知属性页属性调整），所以必须手工调用 Modified 方法。Modified 方法调用的结果是使属性页的标准按钮——应用（Apply）按钮可选，而只有此时单击“应用”按钮或“确定”按钮才能导致 UpdateObject 函数调用使 ActiveX 控件属性更新。

在 UpdatePropertyPage 和 UpdateObject 函数中调用了 TPropertyPage 类的 OleObject 属性，该属性声明为 Variant 变体类型，它对应着现实 ActiveX 控件对象，通过对 OleObject 的访问获得或设置 ActiveX 控件的属性。

图 16-20 展示了一个控件的属性页。

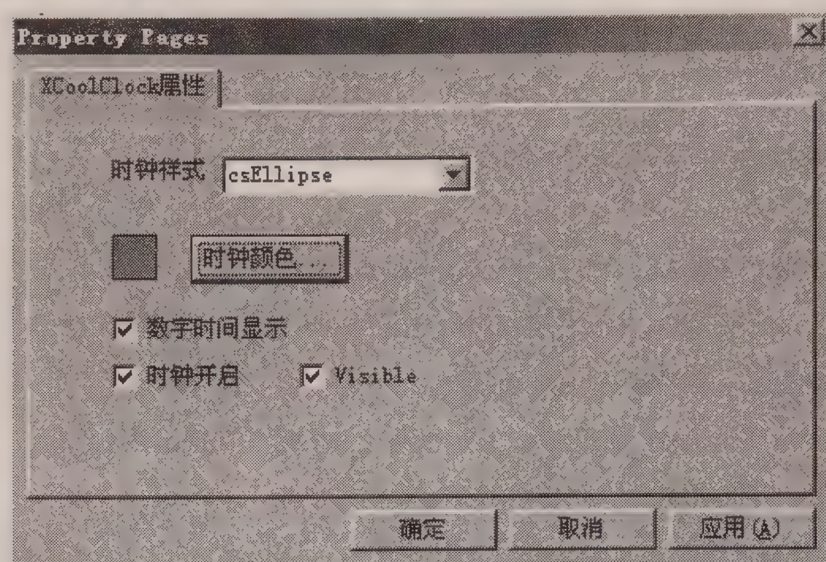


图 16-20　在 Visual Basic 中打开 XCoolClock 控件属性页

最后，要把属性页与 ActiveX 控件联系起来，这就要在 ActiveX 控件实现单元中的 BEGIN_PROPERTY_MAP 和 END_PROPERTY_MAP 之间加一个 PROP_PAGE 宏，并且传递属性页的 GUID 作为参数。属性页的 GUID 是由 C++ Builder 自动分配给属性页的，可以在属性页的单元文件中找到这个 GUID，如 Class_PropertyPage1、Class_PropertyPage2。

```
BEGIN_PROPERTY_MAP(TCoolClockXImpl)
 PROP_PAGE(CLSID_PropertyPage1)
END_PROPERTY_MAP()
```

另外 C++ Builder 提供了四种标准的属性页：Class_DColorPropPage、Class_DfontPropPage、Class_DPicturePropPage、Class_DStringPropPage，这四个标准的属性页分别用于编辑颜色、字体、图像和字符串列表。

16.7.4　ActiveForm 方法

另一种简单的创建 ActiveX 控件的替代方法是使用 ActiveForm，这项技术借鉴于 Visual Basic，在我们需要创建复合组件时 ActiveForm 方法显得特别有意义。

本小节将快速创建一个小型的图像浏览器控件，以演示 ActiveForm 技术的强大功能。

使用 ActiveForm 方法创建 ActiveX 应用，应首先选择【**File/New**】菜单打开 New Item 对话框，在 ActiveX 页中选择 ActiveForm 项，随后在 ActiveForm Wizard 对话框中填写必要的信息后，C++ Builder 将自动生成 ActiveForm 应用框架，如图 16-21 所示。

之后的工作与创建普通的 C++ Builder 应用非常相似，可以可视化的设计窗体，并在事件的响应函数中编制处理代码。不同之处在于 ActiveForm 方法处理的窗体继承于 TActiveForm 类，并且具有特殊的 ActiveForm 接口。窗体类的声明如下：

```
class TViewImgX : public TActiveForm
```

利用这个 TActiveForm 类的“容器”，可以无所顾忌的使用 VCL 组件，因为这些 VCL 对象都作为 ActiveForm 内部对象使用。唯一有所限制的地方仅仅在于 ActiveForm 这个容器本身，ActiveForm 通过接口属性使其作为一个控件而非窗体，具体来说，它将 TForm 类原有

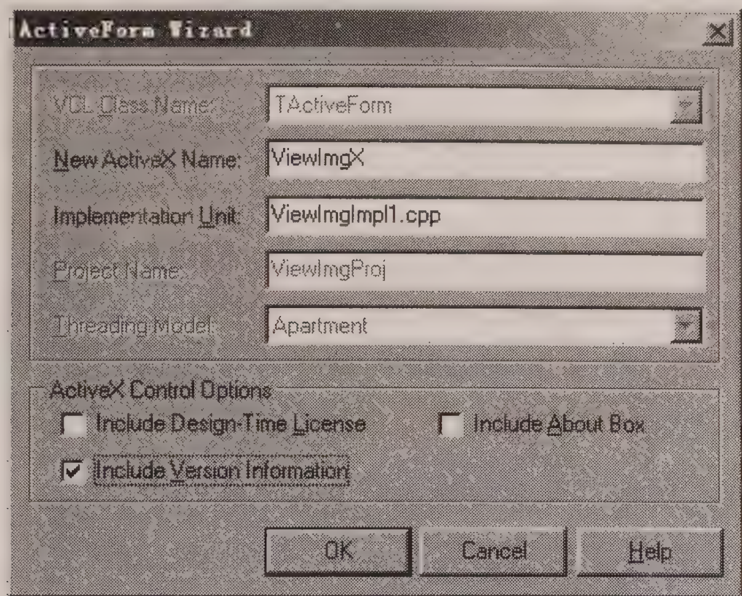


图 16-21　ActiveForm Wizard 对话框

的事件和外部的 ActiveX 事件对应起来，并像一个控件那样输出自己的方法和属性。而如果要用到某个 VCL 组件的属性、方法和事件，也只能在 ActiveForm 的名义下输出。

明确了 ActiveForm 技术创建控件的原理后，回到 ViewImgX 控件的范例中。在 TActiveForm 类的窗体上，简单的放置一些 VCL 组件，如图 16-22 所示。

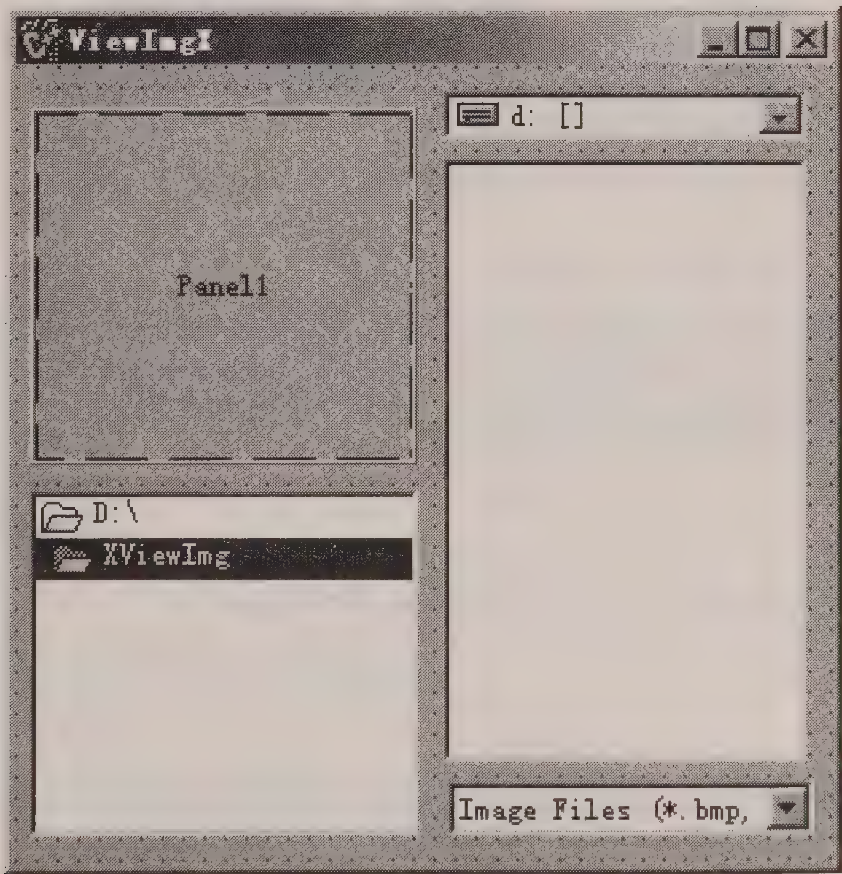


图 16-22　设计 ActiveForm 窗体

设置几个文件浏览组件，使之相互联系。然后响应 FileListBox1 的 OnClick 事件，将选择的图像在 Image1 组件中显示出来：

```
void __fastcall TViewImgX::FileListBox1Click(TObject *Sender)
{
```

```
        Image1->Picture->LoadFromFile(FileListBox1->FileName);
}
```

此外，可以设置 ActiveForm 的 AxBorderStyle 属性，将其设置为 afbSunken，使 ActiveX 控件的边框具有下凹效果。

现在编译工程，就完成了一个具有图像浏览功能的 ActiveX 控件。但是这个控件的功能仅仅是单方面的——它给用户提供浏览图像的功能，但应用程序却对控件一无所知。下面为这个 ActiveForm 控件创建一个简单的属性 ImgName，用于设置或返回显示图像的文件名。图 16-23 是一个应用程序使用控件的原则。

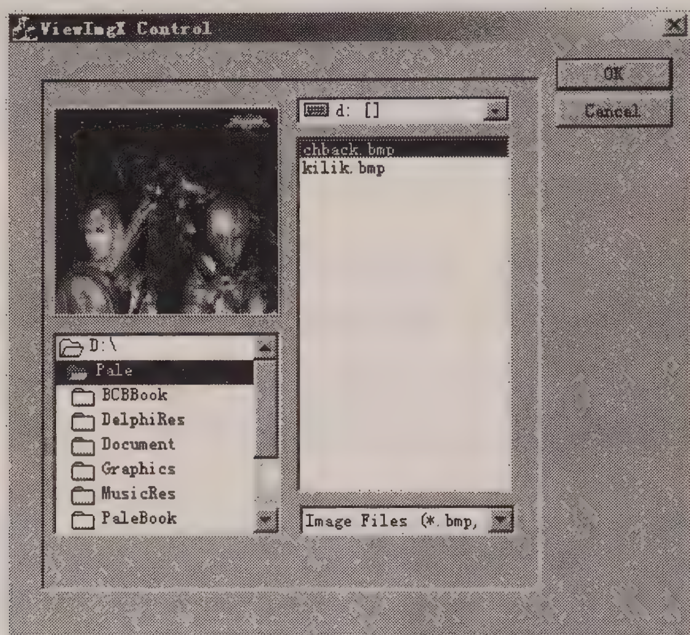


图 16-23　在 VC++应用程序中使用 ViewImgX 控件

打开类型库编辑器，增加属性 ImgName，定义类型为 BSTR（BasicString），然后完成属性读写方法的代码：

Source 为 ActiveForm 控件增加新属性

```
//---------------------------------------------------------------------------
STDMETHODIMP TViewImgXImpl::get_ImgName(BSTR* Value)
{
    try
    {
        *Value = WideString(m_VclCtl->FileListBox1->FileName).Copy();
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_IViewImgX);
    }
    return S_OK;
};
```

```
//--------------------------------------------------------------------------
STDMETHODIMP TViewImgXImpl::set_ImgName(BSTR Value)
{
    try
    {
        const DISPID dispid = 15;
        if (FireOnRequestEdit(dispid) == S_FALSE)
            return S_FALSE;
        m_VclCtl->FileListBox1->FileName = AnsiString(Value);
        m_VclCtl->Image1->Picture->LoadFromFile(AnsiString(Value));
        FireOnChanged(dispid);
    }
    catch(Exception &e)
    {
    return Error(e.Message.c_str(), IID_IViewImgX);
    }
    return S_OK;
};
//--------------------------------------------------------------------------
```

特别应注意的是，在创建 ActiveX 控件字符串类属性时应使用 BSTR，而不应使用字符串或字符串指针，因为只有 BSTR 类型字符串可以在 OLE 对象中使用。

新创建的属性页如图 16-24 所示。

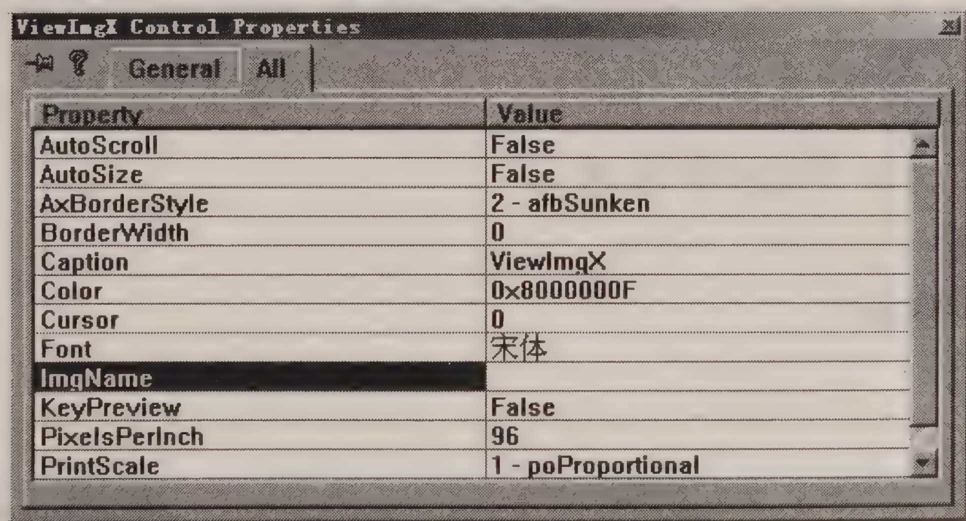
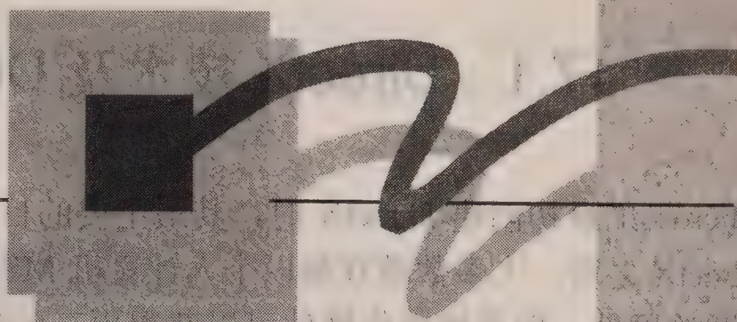


图 16-24　在 VC++中的控件属性页出现了新创建的属性

第17章

DirectX 下的 Block 游戏
——DirectX 编程

DirectX 技术概述
DirectDraw 高性能动画编程
创建基于 DirectX 的游戏程序
Direct3D 编程
DirectSound 编程



DirectX 是一个极佳的游戏程序开发接口，DirectX API 基于 COM 建立，并且相当复杂和庞大，其中包括 2 维、3 维图像，声音，输入设备和网络互联等多项技术。本章将创建一个真正基于 DirectX 技术的游戏程序——Block 游戏，这是一个声光效果俱佳的接球游戏。通过该实例将详细讲解 DirectX 技术中最为重要的部分——DirectDraw 的编程问题。

另外，本章还将对 DirectX 的其他一些部分，包括应用 Direct3D 和 DirectSound 技术编程进行介绍。

本章内容包括:

- DirectX 技术和 DirectX 编程的概括性介绍。
- 如何使用 DirectDraw 编写高性能动画程序。
- 利用 DirectDraw 创建范例程序——Block 游戏，通过实例的创建讲述应用 DirectDraw 编程的各种问题。
- DirectX 技术的其他内容，包括使用 DirectSound、Direct3D 编程。

本章导读

17.1　DirectX 技术及 DirectX 编程概述

当 Microsoft 最初创建 DirectX 时，其最主要的目的是提倡将游戏向 Windows 操作系统发展。在 DirectX 之前，大多数游戏都以个人计算机 MS-DOS 为操作系统。开发这些游戏必须面对各种不同的硬件，但直接操作硬件可以获得更高的性能。而有了 DirectX，游戏开发者便可得益于与硬件无关性，同时仍然能直接访问操作硬件。DirectX 另一个主要目的是提供一个现今使用 MS-DOS 下的应用程序可移动访问的特性，用于更好的访问利用基于 MS-DOS 下的应用程序，并消除个人电脑上的硬件障碍。

DirectX 技术提供了创建基于 Windows 的高性能应用程序方法，即时访问操作现在及将来可用的一切计算机硬件。DirectX 提供应用程序与硬件间坚实可靠的接口操作，减轻安装设置及体现硬件优越性能的复杂程度。使用 DirectX 提供的接口，软件开发者即使在不了解硬件详细资料的状况下也能充分发挥硬件特性。

高性能的 DirectX 游戏可以轻松实现以下技术：

- 发挥特殊优化的加速卡的硬件性能。
- 即插即用设备及其他 Windows 硬件。
- 高效的 Windows 内建的通信服务（DirectPlay）。

17.1.1　DirectX 的组成

现在 DirectX 已经发展到 6.X 版本，并且 DirectX 7.0 也即将发布。DirectX 的家族越来越庞大，包含的技术也越来越多，但是，DirectX 包含 5 个基本的，同时也是最为重要和常用的组成部分，它们是：

- DirectDraw
- Direct3D
- DirectSound
- DirectInput
- DirectPlay

DirectDraw 是与显示硬件相关的接口。它允许开发人员作一些事情，如设置图形模式、改变调色板颜色和操作 2D 图形。通过直接写屏技术和硬件加速大幅度提升 2D 图形显示性能，提供极快的位图移动及多页缓冲。

Direct3D 建立在 DirectDraw 基础上，它包含高级别的保留模式（Retained Mode）和过渡模式(Intermediate Mode)两个 API 子集。使用 Direct3D 能够容易的实时显示和显示复杂的 3D 模型。

DirectSound 提供高性能的声音播放接口，并实现了硬件及软件的实时混音。唯一遗憾的是 DirectSound 没有提供 MIDI 接口，这似乎是 Microsoft 的一个疏忽。

DirectInput 提供游戏输入设备的控制。这个 API 的最大好处在于兼容性，它基于现在及将

来的基于 Windows 的硬件输入 API 及驱动程序之上。现已支持游戏杆，鼠标，键盘及力反馈设备。

DirectPlay 提供创建有多个参与者的游戏和交互式应用的功能，使编程人员可以非常容易的创建通过直接串行连接、Modom、IPX 和 Internet 的联机对战游戏程序。

DirectX 的主要组成部分如图 17-1 所示。

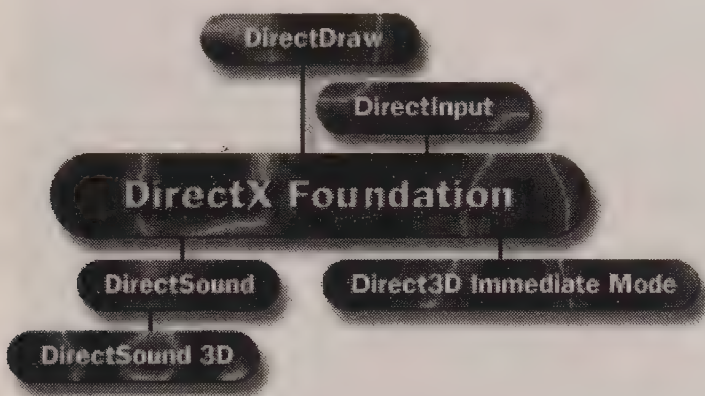


图 17-1　DirectX 的主要组成部分

目前的 DirectX 版本中还包括了其他的一些内容。例如 DirectSetup，提供 DirectX 一次性安装程序；DirectMusic，提供 DirectX 的数字音乐处理；DirectShow，提供 DirectX 视频播放。现在，DirectX 的触角已经涉及到了游戏编程的各个方面，由于 Microsoft 给 DirectX 增加了愈来愈多的功能，它开始超越 DOS 的灵活性，成为多媒体和游戏开发 API 中的绝对领先者。公正地说，Microsoft 的 DirectX 确实是一个非常出色的产品。

17.1.2　DirectX 编程方式

在 C++ Builder 中调用 DirectX API 应该首先加入 DirectX 的相应头文件，例如，如果需要使用 DirectDraw API，那么在程序中应该加入 ddraw.h 头文件。这个头文件是由 Microsoft 公司提供的，它包含在 DirectX SDK 中，实际上， C++ Builder 已经包含了 DirectX 的所有头文件，包括 ddraw.h、d3d.h、dsound.h 等等，这些文件在 C++ Builder 目录的 Include 子目录下。

大多数 DirectX API 都是以组件对象模型（COM）为标准而创建的。调用 DirectX API 需要访问各种 COM 对象，这些对象就像一个黑盒子记录了硬件及应用程序通过一个接口所进行的通信。命令送到或接收自一个通过组件对象模型接口的对象，例如在 DirectDraw 中的 DirectDraw 对象，它可以通过 IDirectDraw2 接口进行各种显示操作。

在 DirectX 编程中，在进行实际的操作之前，首先要做的是创建对象。完成这项工作通常通过一个 Create 函数，例如创建 DirectDraw 对象的函数是 DirectDrawCreate。另一方面，由于 DirectX API 基于 COM，也可以通过标准的 COM 调用来创建对象。这是通过 CoCreateInstance 函数和 Initialize 函数来实现的。

当对象使用完毕之后，需要使用 Release 函数释放一个对象。这也是一个 COM 的标准调用。

在说明了以上问题后，您应该了解，虽然 Microsoft 作了很多有益的工作，使得使用

DirectX API 尽量简单，但事实是，使用 DirectX 进行绘图、输入控制、播放声音等工作比起使用基本的 Windows 方法要复杂的多。但 DirectX 仍然是使程序员们兴奋无比的技术，因为这并非仅仅是完成或者实现某项功能的问题，而是性能问题。

17.2 使用 DirectDraw

DirectDraw 提供了创建 2D 图形所需要的全部功能的 API。无论从哪一个角度来看，DirectDraw 都是 DirectX 中最为基本也是最为有用的一项技术。应该说，对于其他方面，例如音频、输入控制等，Windows 的基本方法（如 MCI、键盘鼠标消息）还是能够满足需要的。但在视频方面，想要创建快速的、高品质的动画，Windows GDI 往往成为了致命的瓶颈。

17.2.1 强劲的 DirectDraw 技术

DirectDraw 是 DirectX API 的一部分。它能让编程人员直接操作显存，硬件显存块移动（Blitter），支持硬件屏遮（OverLay），以及切换平面（Flipping Surface）。

DirectDraw 的主要目的就是使程序员在需要时能够直接写视频缓存，它也支持程序员对视频偏移缓冲区操作。该缓冲区的内容可以迅速转移到可见的视频缓存中，也就是说，采用了双缓冲区（Double Buffer）的技术。这种技术可以用来创建平滑的高速动画。

DirectDraw 自动提供了硬件加速性能的访问。这使得对硬件并不了解的程序员也能毫不费力的发挥出硬件的优势。这就意味着，DirectDraw 可以在程序员毫不知情的情况下最大限度的提高性能。最为明显的一个例子是 Intel 公司的 MMX 技术，即使是在 MMX 技术出现之前编写的 DirectX 程序，现在也能够自动利用 MMX 技术加速视频，因为最新的 DirectDraw 驱动程序已经支持了它。同时，DirectDraw 的硬件抽象层（Hardware Abstraction Layer<HAL>）为直接操作显示硬件提供了一个坚实的接口，并给予最大的优化。

DirectDraw 基于 Windows，可充分发挥操作系统提供的 32-Bit 内存寻址及线性内存模式。DirectDraw 将显存及内存看作为一个大的块，并不是段。

最后，DirectDraw 突破了 Windows GDI（包括建立在 WindowsGDI 基础上的 VCL 图形）的限制。在使用 DirectDraw 技术编写程序时，我们更多的使用独占技术，也就是控制整个屏幕而并非在 Windows GDI 中仅仅是在某一个窗口上绘图。同时，程序员还可以根据需要自由的设置显示分辨率和颜色深度。总之，在 DirectDraw 中 可以控制在 Windows GDI 中可望而不可及的一切。

17.2.2 建立简单的 DirectDraw 程序

以上简单介绍了 DirectX 和 DirectDraw 的一些概念，下面要进行的是具体的操作，编写一个简单的 DirectDraw 程序。并通过程序的创建讲述使用 DirectDraw 编程的方法。

这里完成的简单 DirectDraw 程序通过一个 TTimer 组件不断的在两个 DirectDraw 平面间来

回切换。两个 DirectDraw 平面上分别写着不同的文本。实际上，这个程序和 C++ Builder 提供的 DirectDraw 的第一个例程 DDraw1a 效果几乎完全相同，但这里我们使用了新的 IDirectDraw2 接口，该接口需要 DirectX 5.0 以上版本支持。

处于设计期间的 DirectDraw Demo 程序如图 17-2 所示。

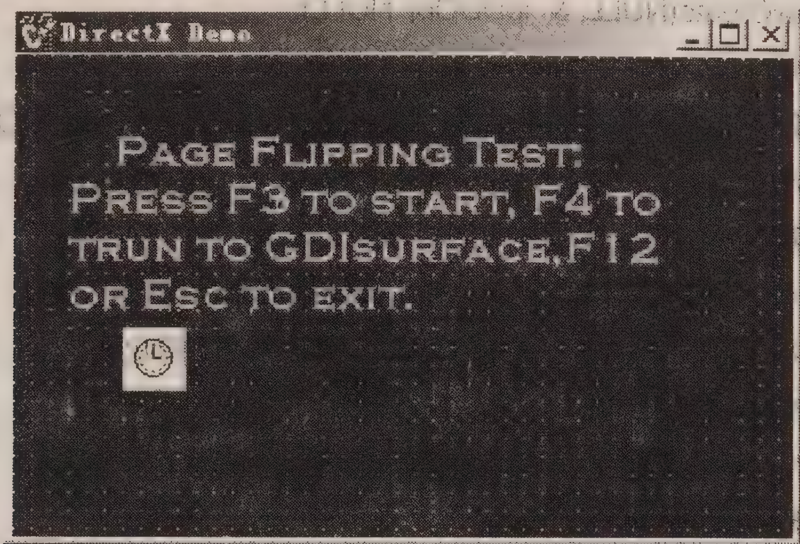


图 17-2　设计期间的 DirectDraw Demo 程序

以下是这个程序的主体部分。

Source 简单的 DirectDraw 例子

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
    lpDDObj = NULL;
    phase = 0;
    bActive = false;
    FrontMsg = "Front buffer (F12 or Esc to quit)";
    BackMsg = "Back buffer (F12 or Esc to quit)";
}
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    lpSPrimary->Release();
    lpSPrimary = NULL;
    lpDDObj->Release();
    lpDDObj = NULL;
}
void __fastcall TForm1::Initialize()
{
    HRESULT ddrval;
    DDSURFACEDESC ddsd;
```

```
DDSCAPS ddscaps;
HDC DC;
char buf[256];
Label1->Visible=false;
ddrval = DirectDrawCreate(NULL, &_lpDDObj, NULL);
if(ddrval == DD_OK) {
        ddrval=_lpDDObj->QueryInterface(IID_IDirectDraw2, (void **)&lpDDObj);
        _lpDDObj->Release();
        if (ddrval==DD_OK) {
                ddrval = lpDDObj->SetCooperativeLevel(Handle,
                        DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN);
                if(ddrval == DD_OK) {
                        ddrval = lpDDObj->SetDisplayMode(640,480, 8,0, 0 );
                if(ddrval == DD_OK) {
                        ddsd.dwSize = sizeof(ddsd);
                        ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
                        ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                                DDSCAPS_FLIP | DDSCAPS_COMPLEX;
                        ddsd.dwBackBufferCount = 1;
                        ddrval = lpDDObj->CreateSurface(&ddsd, &lpSPrimary, NULL);
                if(ddrval == DD_OK) {
                        ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
                        ddrval = lpSPrimary->GetAttachedSurface(&ddscaps,
                                &lpSBack);
                if(ddrval == DD_OK) {
                if (lpSPrimary->GetDC(&DC) == DD_OK) {
                        SetBkColor(DC, RGB(0, 0, 255));
                        SetTextColor(DC, RGB(255, 255, 0));
                        TextOut(DC,0,0,FrontMsg.c_str(),FrontMsg.Length());
                        lpSPrimary->ReleaseDC(DC);
                }
                if (lpSBack->GetDC(&DC) == DD_OK) {
                        SetBkColor(DC, RGB(0, 0, 255));
                        SetTextColor(DC, RGB(255, 0, 0));
                        TextOut(DC,0,0,BackMsg.c_str(),BackMsg.Length());
                        lpSBack->ReleaseDC(DC);
                }
                Timer1->Enabled = true;
                bActive = true;
                return;
```

```
            }
          }
        }
      }
    }
  }
  wsprintf(buf, "Direct Draw 初始化失败 (%08lx)\n", ddrval);
  MessageBox(Handle, buf, "错误", MB_OK);
  Close();
}
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
   TShiftState Shift)
{
  switch (Key)
  {
  case VK_F3:
      if (!bActive) Initialize();
      break;
  case VK_ESCAPE:
  case VK_F12:
      Close();
      break;
  }
}
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
  HDC DC;
  if (lpSBack->GetDC(&DC) == DD_OK)
  {
      if(phase) {
          SetBkColor(DC, RGB(0, 0, 255));
          SetTextColor(DC, RGB(255, 255, 0));
          TextOut(DC, 0, 0, FrontMsg.c_str(), FrontMsg.Length());
          phase = 0;
      }
      else {
          SetBkColor(DC, RGB(0, 0, 255));
          SetTextColor(DC, RGB(0, 255, 255));
          TextOut(DC, 0, 0, BackMsg.c_str(), BackMsg.Length());
          phase = 1;
```

```
            }
            lpSBack->ReleaseDC(DC);
        }
        lpSPrimary->Flip(NULL, 0);
    }
```

这个程序会首先在窗口中显示一段信息，并等待用户的键盘命令。当用户按下 F3 时，事件响应函数会调用 Initialize 函数创建 DirectDraw 以及相关对象，并且切换到 DirectDraw 全屏模式。之后，TTimer 组件启动，在 OnTimer 事件响应函数中，进行 DirectDraw 平面的切换，交替显示不同的信息。

下文将进一步解释这个程序，并借以讲述使用 DirectX 编程的基本步骤。

17.2.3 DirectDraw 编程问题

DirectDraw Demo 程序涉及了 DirectDraw 编程的几个基本要点，包括 DirectDraw 初始化、屏幕模式设置、创建 DirectDraw 平面、使用 GDI 工作、交换 DirectDraw 平面和释放 DirectDraw 对象。

1. DirectDraw 初始化

DirectDraw 的初始化过程包括各种对象的初始化，例如 DirectDraw 对象、DirectDraw Surface 对象、DirectDrawClipper 对象等等。其中 DirectDraw 对象是必不可少的。

创建一个 DirectDraw 对象，应该调用 DirectDrawCreate 函数，该函数的声明是这样的：

```
HRESULT DirectDrawCreate( GUID FAR *lpGUID, LPDIRECTDRAW FAR *lplpDD, IUnknown FAR *pUnkOuter );
```

该函数的第一个参数一般为 NULL，除非想要指定 DirectDraw 只使用硬件（HAL）或者只使用仿真（HEL）。但这往往没有意义，因为 DirectDraw 会自动根据硬件性能优化决定使用 HAL 还是 HEL。除非希望强制依赖硬件特性工作，这时应该传递 DDCREATE_HEADWAREONLY，但如果硬件不能执行，则会导致失败，并且函数返回 DDERR_UNSUPPORTED；如果只使用仿真的话，就传递 DDCREATE_EMULATIONONLY 参数。

第二个参数返回一个创建好的 DirectDraw 对象实例。在上面的示例程序中，我们声明了一个 LPDIRECTDRAW 类型的 DirectDraw 对象变量_lpDDObj，通过参数传递，该变量可以获得 DirectDraw 对象实例。

第三个参数目前没有任何用处，应该设置此参数为 NULL，其他任何值都会报错。这个参数用于将来的 COM 集合扩展。

如果 DirectDrawCreate 调用失败，它会返回某一个 DDERR 值，如果成功，该函数会返回 DD_OK。

如果上述基本对象建立成功，我们可以继续建立新的接口对象以增进性能。通过使用标准 COM 方式——调用 QueryInterface 函数建立新对象。在 IDirectDraw2 接口对象建立后，调用 Release 函数释放_lpDDObj 对象。

2. 屏幕模式设置

在获得 DirectDraw 对象后，下一步工作是设置协作层（Cooperative Levels）。在程序中通过 SetCooperativeLevel 函数调用来完成这项任务。

协作层描述了 DirectDraw 如何与显示相互作用及一些影响显示的因素。在多数情况下，使用 DirectDraw 协作层来决定程序运行于全屏独占模式或运行于窗口。另外，DirectDraw 协作层还有以下功能：

- 允许 DirectDraw 使用 Mode X 解析度，详见 Mode X 及 Mode 13 显示模式。
- 防止使用者用 Ctrl+Alt+Del 重新启动（仅在独占模式）。
- 允许 DirectDraw 响应最大及最小化事件。

应用设置 DirectDraw 协作层的函数 SetCooperativeLevel 声明为：

```
HRESULT SetCooperativeLevel( HWND hWnd, DOWRD dwFlags );
```

hWnd 参数用于保持和一个窗口的联系，dwFlags 参数用于设置协作层。通常使用 DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN 将协作层设置为全屏独占模式。在这种模式下，可以完全使用硬件的一切，包括设置使用定义或动态调色板，改变显示分辨率及进行页交换。也可以设置为 DDSCL_NORMAL，将协作层初始化为窗口模式。

如果设置了全屏独占模式，进一步，可以使用 SetDisplayMode 函数设置分辨率和颜色深度。例如示例程序中：

```
ddrval = lpDDObj->SetDisplayMode(640,480, 8,0,0);
```

它将屏幕分辨率设置为 640×480，颜色深度设置为 8 位（即同显 256 色），用于设置屏幕刷新率的参数设为 0，表示不作修改。

值得注意的是，在 IDirectDraw 接口中只有三个参数，它不支持刷新率的设置。请参看本章实例程序代码。

3. 创建 DirectDraw 平面

当协作层和显示模式设置成功后，下一步的任务是获取两个平面，以便在平面之间进行页面交换。当然 DirectDraw 也允许创建多层平面，设置许多平面，就象艺术家的纸一张一张的层叠，第一个平面称为主平面，而后面的平面称为屏后缓冲。程序事先修改屏后缓冲的图象，然后翻开主平面，屏后缓冲便显示出来了。当系统显示图像，程序继续写屏后缓冲。这样不停继续，就实现了高速高效显示动画。

程序中我们分别通过 CreateSurface 函数和 GetAttachedSurface 函数创建一个主平面和一个后台缓冲平面，这样就实现了双缓冲技术。

CreateSurface 函数需要引用一个 DDSURFACEDESC 类型的变量，而 GetAttachedSurface 函数需要引用一个 DDSCAPS 类型的变量。DDSURFACEDESC 结构和 DDSCAPS 结构都相当复杂，它们处理了一大堆的常量。例如 DDSURFACEDESC 结构声明如下：

```
typedef struct _DDSURFACEDESC
{
   DWORD      dwSize;    // size of the DDSURFACEDESC structure
   DWORD      dwFlags;   // determines what fields are valid
   DWORD      dwHeight;  // height of surface to be created
```

```
DWORD       dwWidth;   // width of input surface
union
{
    LONG        lPitch;         // distance to start of next line (return value only)
    DWORD dwLinearSize;  // Formless late-allocated optimized surface size
};
DWORD       dwBackBufferCount;          // number of back buffers requested
union
{
    DWORD    dwMipMapCount; // number of mip-map levels requested
        DWORD        dwZBufferBitDepth;      // depth of Z buffer requested
        DWORD        dwRefreshRate; // refresh rate (used when display mode is described)
};
DWORD                dwAlphaBitDepth;        // depth of alpha buffer requested
DWORD                dwReserved;             // reserved
LPVOID        lpSurface;          // pointer to the associated surface memory
DDCOLORKEY      ddckCKDestOverlay;  // color key for destination overlay use
DDCOLORKEY      ddckCKDestBlt;          // color key for destination blt use
DDCOLORKEY      ddckCKSrcOverlay;       // color key for source overlay use
DDCOLORKEY      ddckCKSrcBlt;           // color key for source blt use
DDPIXELFORMAT          ddpfPixelFormat;  // pixel format description of the surface
DDSCAPS              ddsCaps;   // direct draw surface capabilities
} DDSURFACEDESC;
```

这个结构相当的复杂，但幸好一般无需填写所有的字段，并且能够从字面明白大多数字段的含义，同时我们可以参看 Direct SDK 来进一步明确这些字段。

4. 使用 GDI 工作

在示例程序中，调用 Windows GDI 函数向 DirectX 屏幕写了一些文本。因为 DirectDraw 没有 API 直接支持文字的输出，所以使用 Windows GDI 工作是必要的。

DirectDraw 的一个重要的优点就是可以与 Windows GDI 相容，如图 17-3 所示。该图显示了 DirectDraw 的工作方式，可以看到 DirectDraw 对象与 GDI 并排存在，并且各自有操作硬件的设备操作层。

在程序中首先使用 GetDC 函数获得后台缓冲平面和主平面的 HDC，然后调用 GDI 函数 TextOut 输出一行文本，同理，也可以使用各种各样的标准 GDI 调用。

5. 切换缓冲平面

交换平面是产生快速动画的关键部分。使用 Filp 函数（主平面对象的一个方法）可以完成缓冲平面的切换工作。

调用 Flip 函数使得主缓冲平面和后台缓冲平面交换，原主平面变成后台平面，原后台平面变成主平面，这样新的内容就会在屏幕上显示出来。

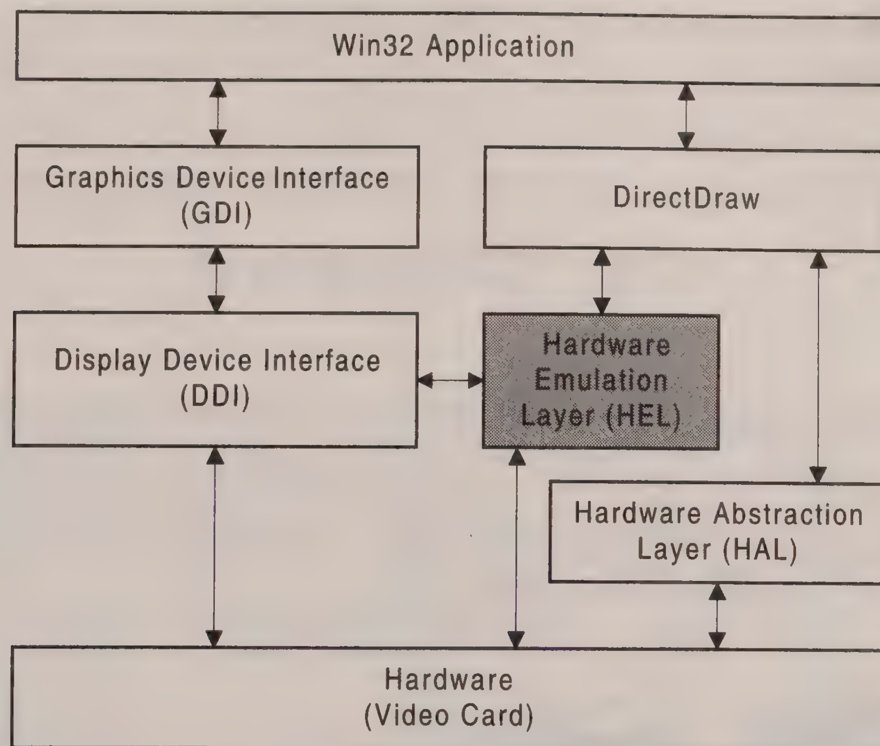


图 17-3 DirectDraw 的工作方式

除了 Flip 函数之外，还有一个重要的切换函数——FlipToGDISurface。该函数是 DirectDraw 对象的一个方法。它可使屏幕切换到 GDI 表面（一般是初始化的那个主平面），如果需要在独占模式下显示一个对话框时，一定要首先调用这个函数，否则，对话框很可能看不到。

可以在上述示例程序的 OnKeyDown 事件响应函数中加入下面的代码，并试着除去 FlipToGDISurface 一行，看看结果如何。

Source 切换到 GDI 平面

```
case VK_F4:
    if (bActive) {
        lpDDObj->FlipToGDISurface();
        Timer1->Enabled=false;
        if (MessageBox(Handle,"This is GDISurface!",
            "DirectDraw 演示",MB_OK)==ID_OK){
            Timer1->Enabled=true;
        }
    }
    break;
```

切换缓冲平面可以说是 DirectDraw 中最为重要的技术，它是 DirectDraw 能够如此之快的关键，它也是双缓冲或多缓冲的核心内容。

6. 释放 DirectDraw 对象

在程序的 OnDestory 事件中，需要销毁所有的 DirectDraw 对象，方法是使用 Release 函

数，并使该对象的变量为一个 NULL 指针，例如：

```
lpDDObj->Release();
lpDDObj = NULL;
```

17.3　实例创建分析

上一节中我们接触了基本的 DirectDraw 编程技术，但 DirectDraw 的威力绝非仅仅如此。现在，我们将创建一个 DirectX 下的 Block 游戏。这个游戏是一个实际的 DirectDraw 应用，它的声光效果非常不错。通过实现这样一个游戏程序，读者将进一步掌握使用 DirectDraw 编程的方法，同时还会从中引入更多的 DirectDraw 话题。

游戏界面如图 17-4 所示，这是一个非常好玩的接球游戏。



图 17-4　Block 游戏的画面

Block 游戏是 Borland 公司例程的“传统项目”。这个游戏要求游戏者控制一个平板接住来回反弹的小球，并通过小球打击各种目标。记得当年的 Turbo Pascal 6.0 中就有一个很不错的 Block 游戏程序（block 一词在此意为阻挡），而在 C++ Builder 4 中包含了一个名为 EarthPong 的游戏例子，这其实也是一个非常简单的 Block 游戏，只是这个程序实在太简单了，它缺省了 Block 游戏的要素——小球没有打击的目标。在此，本书将继续这一传统，利用 DirectDraw 技术创建 Block 游戏的另一版本。

在 Block 游戏中，游戏者操纵一个“平板”接住一个在空间中作直线运动的小球。空间的左、右、上三面都是墙壁，小球在碰到墙壁后会反弹，游戏者可以操纵“平板”在空间的下部横向运动，以阻挡小球掉下。同时，空间中也非空空如也，在经典的 Block 游戏中是很多整齐排列的方砖，当小球碰到方砖时，会将方砖打碎并反弹。经典的 Block 游戏以击碎所有的砖块为最终目标。

在本章创建的 Block 游戏中，将使用随机位置的琉璃彩环代替方砖，击碎不同颜色的彩环会得到相应的分数。每一个彩环都是一个动画物件，它们会以不同的速率摇摆和旋转。

出于兼容性的考虑，在 Block 程序中仅使用了普通的 DirectDraw 接口，而没有使用 IDirectDraw2。实际上，只用修改和增加几行代码就可以利用 DirectX 的新性能。

17.3.1 程序架构

为了使读者能够更好地理解程序，在此将对 Block 游戏程序的架构作一些必要的说明。Block 游戏程序的结构并不十分复杂，但在某些地方，它表现出了一些特殊性。

- 对于彩环这个对象，程序定义了 TRing 类描述该对象。这种做法充分地体现了面向对象编程的优越性。TRing 类包含了彩环对象的各种属性，包括位置、色彩、动画速度等等，最为重要的是，TRing 类实现了一个 Draw 方法，该方法会在 DirectDraw 平面上绘制某一个彩环的下一帧。
- 建立一个彩环对象的列表——TList 类的实例 RingList。该列表的每一项是一个 Tring 对象。通过 RingList 列表，可以方便的管理游戏中出现的彩环。包括彩环的建立、彩环的移除、彩环的遍历等操作。
- DirectDraw 初始化部分，Block 游戏的 DirectDraw 模式初始化为 800×600×256 的全屏独占模式。
- 游戏真正运行期间的代码包含在一个消息处理函数中（下文会具体说明这样做的原因），这个消息是我们自定义的，名为 WM_RUNGAME。
- 程序的运行体，为了加快动画速度，不再使用 TTimer 定时器组件绘制每一帧。当游戏部分开始后，程序将在一个 While 循环中执行，该循环中会调用一个 UpdateFrame 的函数，并且不断的绘制并显示下一帧，而后调用 TApplication 类的 ProcessMessage 函数使程序可以处理其他消息。
- 在 Form 的 OnKeyDown 事件中，如果检测到用户按下了用于退出的键（Esc 或 F12），一个标志量 isActive 将会被设置，而程序运行体的 While 循环是以该标志量为条件，这样动画过程将会退出。

17.3.2 处理位图资源

对于 Block 游戏的位图资源有必要在此说明一下。这个范例程序中的位图资源的特殊性在于，程序用到的所有图像都包含在一个位图中，这个位图截取的一部分如图 17-5 所示。

整个位图资源包含了一副 800×600 的背景图案、三种颜色的彩环（每种彩环都有 60 帧）、以及小球（与 C++ Builder 4 中的 EarthPong 例子程序相同，本程序使用一个地球）和“平板”。将所有的图像放在同一张位图中是有理由的，因为程序在 DirectDraw 中使用的是 256 色的模式，并且使用了调色板。所以，资源位图也必须做成 256 色索引色的位图。如何协调每个物件位图的调色板？最为简便的方法就是将所有的图像（真彩）做在同一位图中，然后再转换为索引色（可以用 PhotoShop 等图像处理工具），这样，所有物件的位图就使用同一个调色板了。



图 17-5　Block 游戏的位图资源，在这个程序中
我们需要对位图资源进行特殊处理

17.3.3　Block 工程说明

Block 工程中包括了下面的一些文件：

- 主窗体单元。文件：Unit1.cpp、Unit1.h。
- 游戏启动窗体单元。文件：Unit2.cpp、Unit2.h。
- 资源文件。文件：Pong.rc。
- Microsoft 提供的 DirectDraw 实用例程。文件：ddutil.cpp、ddutil.h。这两个文件是 Microsoft 随同 DirectX SDK 发行的。如果没有 DirectX SDK，在 C++ Builder 4 的 Examples\Ddraw\utils\目录下也可以找到。

本实例的全部源程序和资源在光盘的\Chapter17\Block\目录下。

17.4　具体实现 Block 游戏

Block 游戏实现了真正的 DirectDraw 动画，同时涉及了另外一些非常重要的 DirectDraw 编程技术，包括在 DirectDraw 中使用位图、创建调色板对象和使用调色板、容错处理、创建透明区域等等。

现在，我们开始着手创建 Block 程序，并从中进一步掌握出色的 DirectDraw 技术。

17.4.1 深入 DirectDraw：调色板和位图对象

在 Block 程序中包含了一个 DirectDraw 初始化函数 Initialize。该函数做了一些和在 DirectDraw Demo 程序中相同的工作，创建一个 DirectDraw 对象、一个主平面对象和一个后台缓冲平面对象。

Source 初始化基本的 DirectDraw 对象

```
ddrval = DirectDrawCreate(NULL,&lpDDObj, NULL);
if( ddrval != DD_OK )
{
    InitFail();
    return;
}
ddrval = lpDDObj->SetCooperativeLevel(
    Handle, DDSCL_EXCLUSIVE|DDSCL_FULLSCREEN);
if( ddrval != DD_OK )
{
    InitFail();
    return;
}
ddrval = lpDDObj->SetDisplayMode( 800,600, 8);
if( ddrval != DD_OK )
{
    InitFail();
    return;
}
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
        DDSCAPS_FLIP | DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
ddrval = lpDDObj->CreateSurface(&ddsd,&lpSPrimary, NULL );
if( ddrval != DD_OK )
{
    InitFail();
    return;
}
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
```

```
ddrval = lpSPrimary->GetAttachedSurface(&ddscaps,&lpSBack);
if( ddrval != DD_OK )
{
    InitFail();
    return;
}
```

现在的问题是，如果需要在 DirectDraw 模式中使用位图和调色板，都要做哪些初始化工作？对于调色板来说，如果需要使用调色板，就必须创建一个 DirectDraw 调色板对象。

1. DirectDraw 调色板

DirectDrawPalette 调色板对象表示一个平面所使用的 16 色或 256 色的调色板索引值。它包含一组平面所使用的由三个数据组成的 RGB 描述色。如果大于 8b 色就毫无必要使用平面的调色板，可以通过调用 CreatePalette 来创建 DirectDrawPalette 对象。

Block 程序中使用位图的调色板如图 17-6 所示。从中可以看出，由于使用了调色板技术，虽然颜色数减少了 2 的 16 次方倍，但画质依然十分出色。

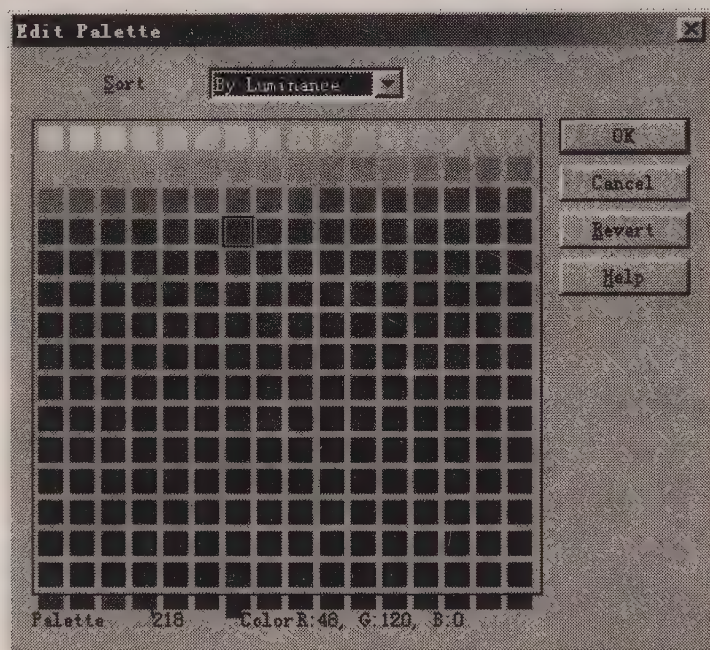


图 17-6　Block 程序中使用位图的调色板

因为 DirectDraw 支持全屏独占模式，这样如果调整系统调色板，并不会因此而出现显示问题或影响其他程序。所以，在 DirectDraw 编程中，使用调色板绝对是首选。这是因为，使用调色板后，画质并不会有明显损失，但却极大的减少了内存资源的占用。

但在 Block 程序中却使用了 Microsoft 提供的实用函数 DDLoadPalette 函数。首先，声明一个指向调色板对象的变量：

```
LPDIRECTDRAWPALETTE lpPal;
```

然后，调用 DDLoadPalette 函数直接从一个位图中获取位图的调色板并创建调色板对象。使用这个函数需要加入 Ddutils.cpp 文件。在创建了调色板之后，调用主平面对象的 SetPalette 方法，通知 DirectDraw 使用调色板。

```
lpPal = DDLoadPalette(lpDDObj,szBitmap);
if (lpPal)
```

```
lpSPrimary->SetPalette(lpPal);
```

DDLoadPalette 的第二个参数指向一个字符串，该字符串标识一个位图资源。

2. 使用位图

另一个问题，如何在 DirectDraw 中使用位图？在 DirectDraw 中，位图也被作为一个缓冲平面对象。这里我们也使用了 Microsoft 提供的实用函数 DDLoadBitmap，这个函数直接从资源中调入一个位图，并创建位图平面对象。

```
lpSBitmap = DDLoadBitmap(lpDDObj, szBitmap, 0, 0);
if( lpSBitmap == NULL )
{
    InitFail();
    return;
}
```

3. 设置透明色

调入的位图包含了背景和精灵（彩环、小球和平板），对于精灵来说，必须透明显示。使用 DDSetColorKey 函数可以直接达到这个目的。例如在本程序中，设置了黑色为透明色。

```
DDSetColorKey(lpSBitmap, RGB(0,0,0));
```

实际上，DirectDraw 可以把某种颜色或某个范围的颜色指定为一个颜色值，这个颜色值是由 DDCOLORKEY 结构即色彩键码说明的，DDCORLORKEY 结构说明如下：

```
typedef struct
    DWORD dwColorSpaceLowValue; // 颜色范围的低端
    DWORD dwColorSpaceHighValue; // 颜色范围的高端
} DDCOLORKEY;
```

设置透明色意味对表面进行拷贝操作时，表面中具有某种颜色的像素不被拷贝。通过 DDCOLORKEY 结构和表面对象的 SetColorKey 方法同样可以设置透明色，如下：

```
DDCOLORKEY ddck;
    ddck.dwColorSpaceLowValue=RGB(0,0,0);
    ddck.dwColorSpaceHighValue=RGB(0,0,0);
lpSBitmap->SetColorKey(DDCKEY_SRCBLT,&ddck);
```

其中 SetColorKey 函数是把色彩键码赋给位图表面 lpSBitmap。这样，在对位图表面的 blt 操作期间 RGB 值为(0,0,0)的像素将不会被拷贝。

17.4.2 构造 TRing 类：绘制 DirectDraw 位图

前文提到，在 Block 程序中，将彩环作为一个类创建。这个类的声明和实现直接写在了主

窗体单元的 Unit1.cpp 和 Unit1.h 文件中。以下是该类的声明：

Source 构建 TRing 类

```
enum TRingType { rtRed,rtGreen,rtBlue};
class TRing : public TObject
{
private:
        int FTop;
        int FLeft;
        int FDelay;
        DWORD FLastTickCount;
        int FCurFrame;
        TRingType FType;
        RECT __fastcall GetRingRect()
        {
                return Rect(FLeft,FTop,FLeft+64,FTop+64);
        }
public:
        __property int Delay = { read = FDelay, write = FDelay };
        void Draw(LPDIRECTDRAWSURFACE lpSBack,
                LPDIRECTDRAWSURFACE lpSBitmap);
        __property TRingType Type = { read = FType, write = FType};
        __property RECT RingRect = { read = GetRingRect };
        __fastcall TRing (int ALeft,int ATop,TRingType AType,int ADelay);
};
```

在此解释一下 TRing 类的属性和私有域。Delay 属性描述彩环对象动画的速率，它的值为彩环每一帧间隔的毫秒数；Type 属性指明彩环对象的类型亦即颜色，有三个可取值 rtRed、rtGreen、rtBlue；RingRect 属性返回彩环对象在屏幕上所占的矩形区域；私有域中 FCurFrame 纪录当前彩环的帧号（注意，每一个彩环都有 60 帧图像）；FlastTickCount 记录上一次换帧的时间。

TRing 类的构造函数很简单，它仅仅获取了对象实例的各个属性值，并初始化了一些私有属性：

Source TRing 类的构造函数实现

```
__fastcall TRing::TRing ( int ALeft,int ATop,TRingType AType,int ADelay)
{
        FLeft = ALeft;
```

```
        FTop = ATop;
        FType = AType;
        FDelay = ADelay;
        FLastTickCount=GetTickCount();
        FCurFrame=0;
    }
```

TRing 类中最为重要的部分是 Draw 对象方法，这个方法将决定彩环的当前帧，并将该帧绘制到后台缓冲平面的指定位置。该函数的实现如下：

Source TRing 类的关键部分——Draw 对象方法的实现

```
    void TRing::Draw( LPDIRECTDRAWSURFACE  lpSBack,
            LPDIRECTDRAWSURFACE  lpSBitmap)
    {
        // 计算彩环的当前帧
        DWORD thisTickCount = GetTickCount();
        if(thisTickCount-FLastTickCount > FDelay)
        {
            FLastTickCount = thisTickCount;
            FCurFrame=(FCurFrame+1)%60;
        }
        RECT        rcRect;
        rcRect.left   = FCurFrame%10*64;
        rcRect.top    = FCurFrame/10*64+600+int(FType)*384;
        rcRect.right  = rcRect.left+64;
        rcRect.bottom = rcRect.top+64;
        lpSBack->BltFast(FLeft,FTop,lpSBitmap,
            &rcRect, DDBLTFAST_SRCCOLORKEY ); // 绘制彩环
    }
```

在这部分代码中可以看到在 DirectDraw 中绘制位图的方法，即使用缓冲平面对象的 BltFast 函数。这个函数用于把某一个缓冲平面上的一块矩形区域绘制到另一个缓冲平面的指定位置。

该函数的前两个参数指定绘制位图的起始位置，第三个参数为指向某一缓冲平面对象的指针，在程序中为指向位图缓冲平面的指针 lpSBitmap，第四个参数是一个指向 RECT 类型的指针，用于指定绘制区域，第五个参数为标志变量，在程序中传递了 DDBLTFAST_SRCCOLORKEY 值表示使用透明色绘制。

另外，在 DirectDraw 中，还支持另外两个 blt 函数，它们是 Blt 和 BltBatch。Blt 函数虽然较 BltFast 函数速度慢，但功能更强，例如支持拉伸、剪切等操作。

17.4.3 游戏的启动部分

从现在开始将进入实质性的游戏流程阶段。在 Block 程序中，我们创建一个启动窗体（如图 17-7 所示），用于让用户完成游戏难度和同时出现彩环数目的设置。

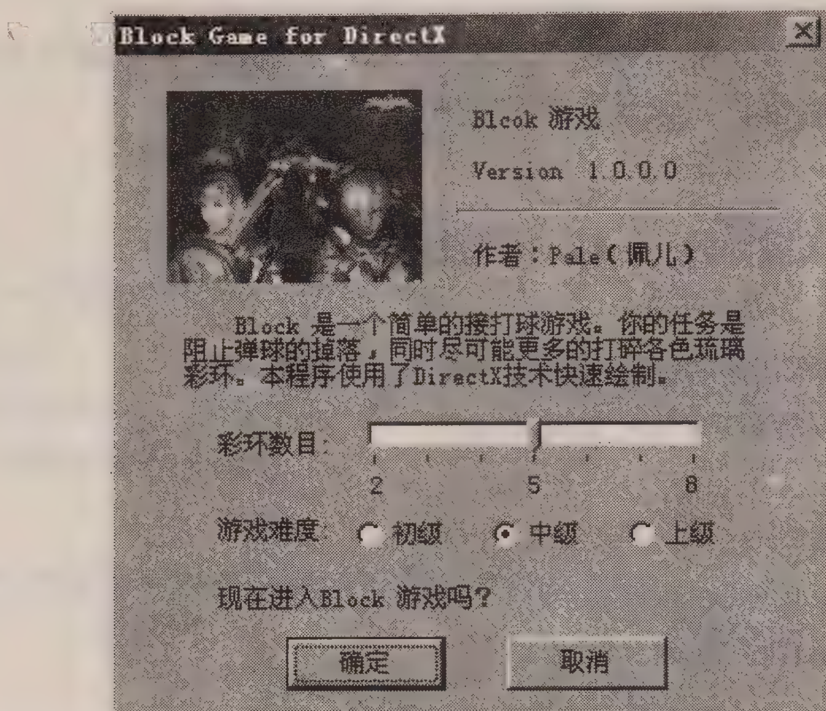


图 17-7 Block 游戏的启动窗体

启动窗体设为 Available Form，可以在主窗体单元中手动创建它。选择在主窗体类的构造函数中创建并显示启动窗体，这样启动窗体就能够先于主窗体显示。下面给出 Block 程序主窗体的构造函数：

Source Block 程序主窗体构造函数

```
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
    Form1->Cursor=crNone;
    Form1->BorderStyle=bsNone;
    TForm2 *Form2=new TForm2(Application);
    isRunApp = true;
    isActive = false;
    RingList = new TList();
    randomize();
    if (Form2->ShowModal()==IDOK){
        if (Form2->RadioButton1->Checked)
            SpeedUp=0;
        else if (Form2->RadioButton2->Checked)
```

```
            SpeedUp=1;
        else SpeedUp=2;
        for (int i=0;i<Form2->TrackBar1->Position;i++)
        {
            RingList->Add(new TRing(random(736),random(300),
            TRingType(random(3)),random(85)+5));
        }
        delete Form2;
    }
    else {
        delete Form2;
        isRunApp = false;
        Application->Terminate();
    }
    LOGFONT fontRec;
    memset(&fontRec,0,sizeof(LOGFONT));
    fontRec.lfHeight = -16;
    fontRec.lfWeight = FW_BOLD;
    lstrcpy(fontRec.lfFaceName,"Copperplate Gothic Bold");
    hFont=CreateFontIndirect(&fontRec);
}
```

可以看到，在主窗体的构造函数中，除了显示游戏启动窗体获得用户设定信息之外，还做了以下几件事：

- 创建一个 TRing 类的列表 RingList。
- 创建数个 TRing 类的实例，并加入 RingList 列表中。
- 创建一个 GDI 字体，用于在 DirectDraw 中使用标准的 GDI 调用输出文字。

在构造函数执行后，将发生 OnCreate 事件，在该事件中，我们调用创建好的 Initialize 函数初始化 DirectDraw。

17.4.4 游戏进行部分

当 DirectDraw 启动之后，屏幕将显示窗体 TLabel 组件的提示信息，这时，Block 程序等待一个键盘消息。

处于设计期间的 Block 程序主窗体如图 17-8 所示。

Source 键盘事件处理函数

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
    TShiftState Shift)
```

```
{
    switch(Key){
    case VK_F3:
        isActive=true;
        EarthX = 200; EarthY = 25;
        DirX = 6; DirY = 6;
        Score = 0;
        PostMessage(Handle, WM_RUNGAME,0,0);
        break;
    case VK_ESCAPE:
    case VK_F12:
        isActive=false;
        Close();
    }
}
```

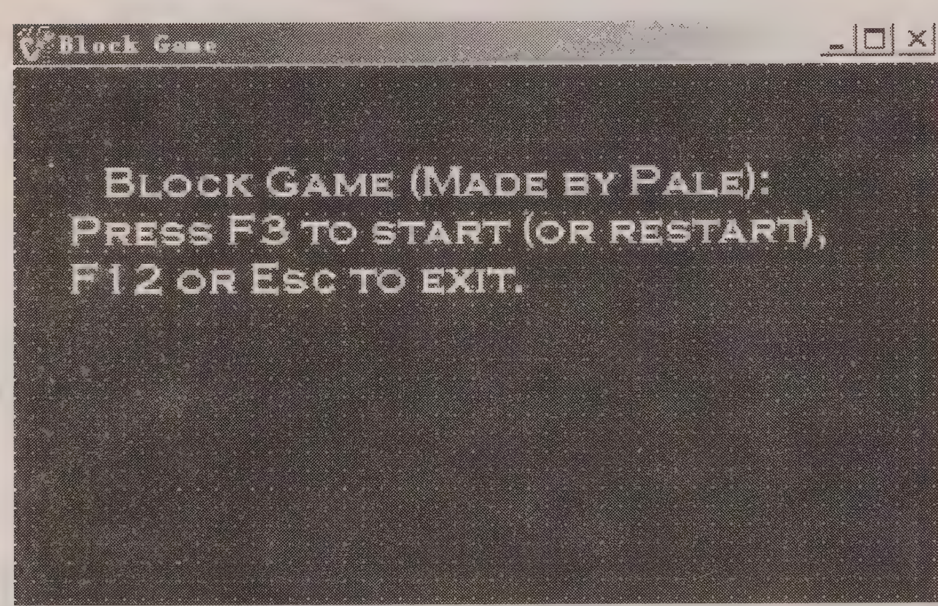


图 17-8　设计期间的 Block 程序主窗体

可以看到，如果用户按下了 F3 键，程序将发送一个自定义消息 WM_RUNGAME 开始游戏的进行，游戏的执行代码是包含在一个消息的处理函数中的。WM_RUNGAME 消息的处理函数 RunGame 如下：

Source 定义消息 WM_RUNGAME 的处理函数

```
void __fastcall TForm1::RunGame(TMessage &Message)
{
    do
    {
        UpdateFrame();
        Application->ProcessMessages();
```

```
    } while(isActive);
}
```

程序会在这个 do-while 循环中不断的被执行，直到游戏结束为止。调用 UpdateFrame 函数推进游戏。同时不断使用 Application 对象的 ProcessMessages 方法，使得程序仍然能够处理消息，包括鼠标消息（用户移动“平板”）和键盘消息（用户发出退出或重新开始的命令）。

UpdateFrame 函数是整个 Block 游戏的关键，请先阅读该函数的代码，稍后将对这部分代码做出解释。

Source UpdateFrame 函数，用于推进游戏并绘制画面

```
void __fastcall TForm1::UpdateFrame()
{
    HRESULT ddrval;
    RECT rcRect;
    rcRect = Rect(0,0,800,600);
    // 绘制背景
    while(true)
    {
        ddrval = lpSBack->BltFast(0,0,lpSBitmap,
            &rcRect, DDBLTFAST_NOCOLORKEY );
        if( ddrval == DD_OK )
            break;
        if( ddrval == DDERR_SURFACELOST )
        {
            if( RestoreAll()!= DD_OK )
                return;
        }
        if( ddrval != DDERR_WASSTILLDRAWING )
        {
            return;
        }
    }
    // 绘制平板
    rcRect=Rect(686,622,765,635);
    lpSBack->BltFast(TableX-39,550,lpSBitmap,
        &rcRect, DDBLTFAST_SRCCOLORKEY );
    // 绘制彩环
    TRing *tmpRing;
    for (int i=0;i < RingList->Count; i++)
```

```
        {
            tmpRing = (TRing*)RingList->Items[i];
            tmpRing->Draw(lpSBack,lpSBitmap);
        }
        // 绘制小球
        DrawEarth();
        // 输出文字
        HDC DC;
        Msg="Score: "+IntToStr(Score);
        if (lpSBack->GetDC(&DC) == DD_OK)
        {
            SelectObject(DC,hFont);
            SetBkMode(DC,TRANSPARENT);
            SetTextColor(DC, RGB(255,255,255));
            TextOut(DC,0,0,Msg.c_str(),Msg.Length());
            TextOut(DC,0,20,Msg1.c_str(),Msg1.Length());
            TextOut(DC,0,35,Msg2.c_str(),Msg2.Length());
            lpSBack->ReleaseDC(DC);
        }
        while(true)
        {
            ddrval = lpSPrimary->Flip(NULL,0);
            if( ddrval == DD_OK )
                break;
            if( ddrval == DDERR_SURFACELOST )
            {
                if( RestoreAll()!= DD_OK )
                    break;
            }
            if( ddrval != DDERR_WASSTILLDRAWING )
                break;
        }
    }
```

绘制背景图像、绘制平板、绘制彩环等工作都是通过 BltFast 函数完成的，这一点在前面已经讨论过，在此不再赘述。

Block 游戏的画面如图 17-9 所示，可以看到画面中同时绘制了多个彩环。

输出文字部分运用了一些 GDI 函数，这段程序调用 SelectObject 函数使用定义好的字体。SetBkMode 函数设置模式使文字能够透明输出，在后台缓冲区绘制完成后，调用 Flip 函数更新屏幕。



图 17-9　Block 游戏的画面

但这里有一个问题，缓冲平面对象的方法，包括 BltFast 和 Flip 函数等，并不一定会正常执行。在这段程序中，演示了如何对缓冲平面对象操作的进行容错处理。

缓冲平面操作的错误可能来源于两个方面——表面丢失和操作未完成。

第一，表面丢失。当用户按下了 Alt+Tab 键离开、并按下 Alt+Tab 键回到程序时，如果正在进行某个操作，该现象发生，操作函数会返回 DDERR_SURFACELOST。这时需要恢复缓冲表面，为此创建一个 RestoreAll 函数完成此功能，这个函数如下：

Source RestoreAll 函数，恢复丢失表面

```
HRESULT __fastcall TForm1::RestoreAll()
{
    HRESULT ddrval;
    ddrval = lpSPrimary->Restore();
    if( ddrval == DD_OK )
    {
        ddrval = lpSBitmap->Restore();
        if( ddrval == DD_OK )
            DDReLoadBitmap(lpSBitmap, szBitmap);
    }
    return ddrval;
}
```

这里的关键在于主平面对象的 Restore 方法和 DDReLoadBitmap 函数。

第二，操作未完成。当前一个缓冲平面操作没有完成时将发生这个错误，操作函数返回

DDERR_WASSTILLDRAWING，解决这个问题可以将代码放在一个 while 循环中，直到缓冲操作可以正常进行才退出循环。

严格来说，每一次调用缓冲平面操作函数都应该使用上述结构进行容错处理。但实际上，上述两种错误的发生概率非常之小（至少笔者和编辑从未遇到过这种情况），所以可以看到，Block 程序中很多平面对象操作都没有使用该结构，而是直接调用了函数。

17.4.5 实现规则

以上已经完成了 Block 游戏总体框架，读者应该已经清楚该程序的运行结构。在 Update 函数中根据精灵的位置状态绘制了画面，现在的问题是，如何控制各个精灵之间的状态变化？这就是一个设计并实现游戏规则的问题。

为了使读者能够更好的理解该程序，在此详细描述一下这个 Block 游戏的规则：

- 小球（地球）沿直线运动，碰到上、左、右墙壁会反弹，反弹遵守入射角等于反射角的规律。
- 小球碰到平板（被接住）也会反弹，这种反弹遵守这样的规则：在平板中线以左碰撞会向左反弹（无论小球来自哪一方向），在平板中线以右碰撞会向右反弹，反射的 X 向速度会根据碰撞点与中线的距离而不同，越偏离中心，X 向速度越大，Y 向速度保持恒值。由于 X 向速度和 Y 向速度的差异，反射会沿不同的角度。
- 当小球碰到琉璃彩环时，该彩环将被击碎，同时小球也会按照规则反弹：如果小球碰到红色彩环，则会沿原路反弹（X 向速度反向、Y 向速度反向），并且 X 向和 Y 向速率都会增加；如果小球碰到绿色彩环，则会使 X 向速度反向，同时 X 向速率增加；如果碰到蓝色彩环，则会使小球 Y 向速度反向，同时 Y 向速率增加。速率增加的多少会根据游戏难度不同而不同，初级难度速率不增加。
- 当某个彩环被击碎，一个新的彩环会随机在另一个地方出现。

对于彩环精灵来说，它的位置是不改变的，只是在固定位置做出动态效果，这点已经在 TRing 类的 Draw 对象方法中实现了。

“平板”的位置控制也较为简单。对于平板精灵来说，它的位置变化由用户的鼠标来控制。在 Form 的 OnMouseMove 函数中实现平板位置变化的处理。

Source 处理鼠标事件，控制“平板”移动

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TShiftState Shift,
    int X, int Y)
{
    TableX=X;
    if (TableX<42) TableX=42;
    if (TableX>758) TableX=758;
}
```

最为复杂的是小球的位置变化处理，DrawEarth 函数实现这项任务。虽然要处理和计算的内容很多，但代码是容易理解的，并且并不涉及技术问题，所以不对这段代码进行详细的解释。

Source 处理小球位置变化函数

```
void __fastcall TForm1::DrawEarth()
{
    RECT rcRect=Rect(647,605,674,632);
    EarthX += DirX; // 小球移动
    EarthY += DirY;
    TRing *tmpRing;
    RECT ringRect;
    int flag=-1;
    // 击碎彩环的处理
    for (int i=0;i < RingList->Count; i++)
    {
        tmpRing = (TRing*)RingList->Items[i];
        ringRect=tmpRing->RingRect;
        if (PtInRect(&ringRect,Point(EarthX,EarthY))) // 是否碰撞
        {
            flag=i;
            switch (tmpRing->Type){
            case rtGreen:
                DirX=-DirX-SpeedUp*DirX/abs(DirX);
                Score+=30;
                break;
            case rtBlue:
                DirY=-DirY-SpeedUp*DirY/abs(DirY);
                Score+=40;
                break;
            case rtRed:
                DirX=-DirX-SpeedUp*DirX/abs(DirX);
                DirY=-DirY-SpeedUp*DirY/abs(DirY);
                Score+=50;
                break;
            }
        }
    }
    if (flag!=-1){ // 击碎了某个彩环
```

```
            sndPlaySound("Glass.wav",SND_ASYNC|SND_FILENAME);
            tmpRing = (TRing*)RingList->Items[flag];
            delete tmpRing;
            RingList->Delete(flag); // 删除被击碎的彩环
            RingList->Add(new TRing(random(736),random(300),
            TRingType(random(3)),random(85)+5)); // 生成新的彩环
        }
        if (EarthY<12) // 碰顶墙
            DirY = -DirY;
        else if (EarthY>=536&&EarthY<=548&&
            EarthX>=TableX-39&&EarthX<=TableX+39)
        {   // 用户接住了小球
            sndPlaySound("Bump.wav",SND_ASYNC|SND_FILENAME);
            Score+=5;
            DirY=-6;
            if (EarthX<TableX){
                DirX=-4+(EarthX-TableX)/6;
            }
            else {
                DirX=4+(EarthX-TableX)/6;
            }
        }
        if (EarthX > 788||EarthX<12) // 碰左右墙
            DirX = -DirX;
        if (EarthY<=580) // 小球没有掉落，绘制小球
            lpSBack->BltFast(EarthX-13,EarthY-13,lpSBitmap,
                &rcRect, DDBLTFAST_SRCCOLORKEY );
        else { // 小球掉落，游戏停止进行
            isActive=false;
            sndPlaySound("Explode.wav",SND_SYNC|SND_FILENAME);
        }
    }
```

这里唯一需要说明的是PtInRect函数，这是一个Windows API函数，它用来判断一个点是否在某个矩形内部。调用这个函数用于判断小球是否击碎了彩环，Block 程序中的判断规则是：如果小球的中心点运动到了某个彩环 64×64 的矩形范围内，则判定为碰到。

17.4.6 最后的工作——释放对象

在实例程序 Block 中，一定要记得释放每一个对象，包括 DirectDraw 对象和彩环对象。这

些工作在 OnDestory 事件响应函数中进行。

Source 在 FormDestory 函数中释放对象

```
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    sndPlaySound("Exit.wav",SND_SYNC|SND_FILENAME);
    TRing *tmpRing;
    for (int i=0;i<RingList->Count;i++) // 释放 Ring 对象
    {
        tmpRing = (TRing*)RingList->Items[i];
        delete tmpRing;
    }
    delete RingList;
    // 释放 DirectDraw 对象
    if( lpDDObj != NULL )
    {
        if( lpSPrimary != NULL ){
            lpSPrimary->Release();
            lpSPrimary = NULL;
        }
        if( lpSBitmap != NULL ){
            lpSBitmap->Release();
            lpSBitmap = NULL;
        }
        if( lpPal != NULL ){
            lpPal->Release();
            lpPal = NULL;
        }
        lpDDObj->Release();
        lpDDObj = NULL;
    }
}
```

这里请注意程序中释放彩环对象的方法，它首先遍历了 RingList 列表，获得每一个存在的 Ring 对象指针，然后释放所有 Ring 对象，最后释放 RingList 对象。

这段程序中释放 DirectDraw 对象的方式是非常标准和严谨的，一般都应使用这样的结构释放 DirectDraw 对象。而我们在本章开始的 DirectDraw 程序中采用的释放方法，虽然简便，但却可能导致出错。

17.5 DirectX 技术的其他部分

在前面花了大量的篇幅讨论了 DirectX 中最为重要的部分——DirectDraw 技术的使用方法，并演示创建了使用 DirectX 技术的游戏。以下将对如何使用其他两种重要的 DirectX 技术——Direct3D 和 DirectSound 做一些概括性的介绍，以帮助读者进一步掌握这些技术。

17.5.1 使用 Direct3D

当我们需要进行一些高性能的 3D 编程时，除了使用 OpenGL，也可以考虑使用 DirectX 中的一部分——Direct3D 来作为完成这项工作的 API。

有一点应该明确，Direct3D 是建立在 DirectDraw 基础上的。Direct3D 必须同 DirectDraw 密切合作来显示 3D 模型和图景。Direct3D 基本上提供了在 3D 模型基础上进行各种操作的所有计算。可以使用 Direct3D 来处理 3D 部分，然后告诉 DirectDraw 要显示什么。

Direct3D 有两种模式：保留模式和过渡模式。保留模式是高层的 Direct3D API，它提供函数支持从.x 格式文件中装载和存入 3D 模型；过渡模式是底层的 Direct3D API。本小节仅就保留模式做一些讨论。

1. 初始化 Direct3D 对象

与 DirectDraw 类似，使用 Direct 3D 也要初始化一些对象。假设想在 C++ Builder 中编写一个应用 Direct 3D 的程序，那么首先要做的工作是创建一个 Direct3D 保留模式对象。下段代码说明如何初始化 Direct3D 保留模式对象。

```
LPDIRECT3DRM lpD3DRMObj = NULL;
if ( Direct3DRMCreate(&lpD3DRMObj)!= D3DRM_OK )
{
    InitFail();
    return;
}
```

这段程序调用了 Direct3DRMCreate 函数，该函数的声明是：

HRESULT Direct3DRMCreate(LPDIRECT3DRM FAR *lplpDirect3DRM);

为了进行 3D 显示，接下来必须创建 Z 缓冲区（Z-Buffer），并且附着在 DirectDraw 后台缓冲区。与创建 DirectDraw 绘制平面相同，使用函数 CreateSurface 来创建 Z 缓冲。常量 DDSD_ZBUFFERBITDEPTH 将告诉 CreateSurface 函数需要设置 Z 缓冲的深度，在 DDSURFACEDESC 结构的 dwZBufferBitDepth 域继续设置该深度。然后，调用 AddAttachedSurface 函数附着 Z-Buffer 平面。

范例程序如下：

Source 在 Direct3D 中创建 Z 缓冲

```
LPDIRECTDRAWSURFACE lpSZBuffer = NULL;
```

```
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT |
        DDSD_WIDTH | DDSD_ZBUFFERBITDEPTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_ZBUFFER | DDSCAPS_SYSTEMMEMORY;
ddsd.dwHeight = 600;
ddsd.dwWidth = 800;
ddsd.dwZBufferBitDepth = 16;
ddrval = lpDDObj->CreateSurface(&ddsd,&lpSZBuffer, NULL );
if( ddrval != DD_OK )
{
        InitFail();
        return;
}
ddrval = lpPrimary->AddAttachedSurface(lpSZBuffer);
if( ddrval != DD_OK )
{
        InitFail();
        return;
}
```

2. 初始化图景（Scene）和设备（Device）

在这一部分中，需要完成以下 3 个步骤：

(1) 使用函数 CreateFrame 来创建一个图景图像，也就是 parent frame（还可以创建一些 child frame）。

(2) 使用函数 CreateDeviceFromSurface 来设定 Direct3D 在什么表面内显示。

(3) 使用 SetQuality 函数来设置 3D 显示质量。

以下是三个关键函数的声明：

```
HRESULT CreateFrame (LPDIRECT3DRMFRAME lpD3DRMFrame,
                    LPDIRECT3DRMFRAME *lplpD3DRMFrame);
HRESULT CreateDeviceFromSurface ( LPGUID lpGUID, LPDIRECRTDRAW lpDD,
LPDIRECTDRAWSURFACE lpDDSBack, LPDIRECT3DRMDEVICE *lplpD3DRMDevice);
HRESULT SetQuality (D3DRMRENDERQUALITY rqQuality);
```

创建图景和设备的示例代码如下。

Source 在 Direct3D 中创建图景和设备

```
LPDIRECT3DRMDEVICE lpD3DRMDevice = NULL;
LPDIRECT3DRMFRAME lpD3DRMScene = NULL;
ddrval = lpD3DRMObj->CreateFrame(NULL,&lpD3DRMScene);
if( ddrval != DD_OK )
```

```
{
        InitFail();
        return;
}
ddrval = lpD3DRMObj->CreateDeviceFromSurface(NULL,lpDDObj,lpSPrimary,
            &lpD3DRMDevice);
if( ddrval != DD_OK )
{
        InitFail();
        return;
}
ddrval = lpD3DRMDevice->SetQuality (D3DRMRENDER_GOURAUD);
if( ddrval != DD_OK )
{
        InitFail();
        return;
}
```

3. 显示 3D 物体

当然可以在 Direct 3D 程序中自己建立 3D 模型并显示出来，但由于篇幅所限，对这部分内容将不做讨论。Direct 3D 直接支持 Microsoft 公司的.x 格式文件。以下说明如何从一个.x 文件中载入一个 3D 模型并显示出来

首先使用 CreateMeshBuilder 函数创建一个网格。然后，调用 D3DRMMeshBuilder 对象的 Load 方法把一个.x 文件调入 MesBuilder。如果调入成功，则可以继续调用 D3DRMMeshBuilder 对象的 SetQuailty 方法设置网的显示质量。最后，调用 ScaleMesh 函数缩放网，以便网格模式可以在屏幕上以合适大小显示出来。

这之后就可以开始着手进行一些更为有意义的工作，如纹理贴图、设置光源、设置观察点等。

在 3D 模型表面绘制纹理需要用到下面几个函数：

LoadTexture

SetTexture

CreateWrap

Apply

LoadTexture 函数用于调入一个作为纹理的.bmp 文件；SetTexture 函数设置这个纹理，使之与 MeshBuilder 相连；CreateWrap 函数创建一个 D3DRMWRAP 对象，并设置包装的区域；最后使用 Apply 函数（D3DRMWRAP 对象的方法），将纹理绘制到 3D 网格上。

创建一个光源需要用到下面两个基本函数：

CreateLightRGB

AddLight

这两个函数的意义很明显：CreateLightRGB 用于创建一个光源对象并且可以设置光源颜

色；AddLight 函数可以为图景添加方向光。

最后，还可以设置图景的观察点，所用的函数是 CreateViewport。

在图景的一切都设置好之后，就可以显示整个 3D 图像了。可以首先使用 Clear 函数来清除观察，就象是清屏一样。然后，使用 Render 函数来显示新观察。这样就可以在屏幕上看到精彩的 3D 图像了。

17.5.2 使用 DirectSound

DirectSound 是一组强大的声音播放 API。声音效果在多媒体及游戏程序中是必不可少的，使用 DirectSound 播放声音有以下三点好处：

- 声音精确到可以在指定的时刻被播放出来。这个特点非常适合于游戏程序的需要，DirectSound 保证声音效果和游戏事件同步。
- 允许在同一时间播放两段音乐，DirectSound 直接支持实时混音输出。例如，在游戏中可以将一段 Wave 作为背景音乐，同时将另一段 Wave 作为音效，使用 DirectSound，两段 Wave 就可以同时播放。而这在 MCI 中是绝不可能的。
- 新版的 DirectSound 还支持 3D 音效。

1. 初始化 DirectSound

使用 DirectSound 编程首先也要建立一个 DirectSound 对象，这是在 DirectX 编程中都要做的第一件事，使用 DirectSoundCreate 函数创建 DirectSound 对象。

下面是 DirectSoundCreate 函数的声明：

```
HRESULT DirectSoundCreate(LPGUID, LPDIRECTSOUND *, LPUNKNOWN);
```

示例代码如下：

```
LPDIRECTSOUND lpDSObj = NULL;
if ( DirectSoundCreate(NULL,&lpDSObj,NULL) != DS_OK )
{
        InitFail();
        return;
}
```

接下来，使用 SetCooperativeLevel 函数设置 DirectSound 的协作层，一般选择 DSSCL_NORMAL 模式。示例代码如下：

```
lpDSObj-> SetCooperativeLevel( Handle, DSSCL_NORMAL);
```

2. 播放声音

使用 DirectSound 播放声音，必须首先将声音文件载入声音缓冲区。事实上，完成这个工作包含了下面的步骤：

(1) 调 Wave 文件到一个临时缓冲区。

(2) 得到一个指向存储在 Wave 文件中的格式信息的指针。

(3) 使用 CreateSoundBuffer 函数创建一个 DirectSound 声音缓冲区。

(4) 锁定声音缓冲区，准备写入数据。

(5) 从临时缓冲区中拷贝声音数据到声音缓冲区。

(6) 解锁 DirectSound 声音缓冲，释放内存中的临时缓冲区。

DirectSound 不提供将 Wave 文件直接调入缓冲区的函数，所以必须自己完成这项工作。但这并非一件容易的事，它需要编程人员对 Wave 文件的格式有所了解。在处理 Wave 文件时可以考虑使用低级多媒体 API 函数，包括 mmioOpen、mmioDescend、mmioRead、mmioFOURCC、mmioClose 等等，因为这些函数为 RIFF 格式提供了内在支持。请参看本书第 7 章中底层多媒体 API 部分。

这里用到了下面几个 DirectSound 函数：

CreateSoundBuffer

Lock

Unlock

CreateSoundBuffer 创建一个缓冲区对象，这个函数需要传递一个 DSBUFFERDESC 结构的变量。Lock 函数用于锁定缓冲区对象，只有在缓冲对象锁定之后，才可以使用 memcpy 或 CopyMemory 将声音数据写入。以下是示例程序：

Source 将声音数据写入 DirectSound 缓冲

```
bool AppWriteDataToBuffer(
    LPDIRECTSOUNDBUFFER lpDSBuffer, DWORD dwOffset,
    LPBYTE lpbSoundData, DWORD dwSoundBytes)
{
        LPVOID lpvPtr1;
        DWORD dwBytes1;
        LPVOID lpvPtr2;
        DWORD dwBytes2;
        HRESULT hr;
        hr = lpDSBuffer->lpVtbl->Lock(lpDSBuffer, dwOffset, dwSoundBytes, &lpvPtr1,
                &dwBytes1, &lpvPtr2, &dwBytes2, 0);
        if(DSERR_BUFFERLOST == hr) {
                lpDSBuffer->lpVtbl->Restore(lpDSBuffer);
                hr = lpDSBuffer->lpVtbl->Lock(lpDSBuffer, dwOffset, dwSoundBytes,
                        &lpvPtr1, &dwAudio1, &lpvPtr2, &dwAudio2, 0);
        }
        if(DS_OK == hr)
                CopyMemory(lpvPtr1, lpbSoundData, dwBytes1);
                if(NULL != lpvPtr2) {
                        CopyMemory(lpvPtr2, lpbSoundData+dwBytes1, dwBytes2);
                }
                hr = lpDSBuffer->lpVtbl->Unlock(lpDSBuffer, lpvPtr1, dwBytes1, lpvPtr2, dwBytes2);
```

```
            if(DS_OK == hr)
                return true;
        }
        return false;
    }
```

当程序成功的将 Wave 文件中的声音数据调入 DirectSound 声音缓冲区后，精彩的事情开始了。首先可以调用 SetCurrentPosition 函数把播放位置设为 0，也就是说，设置位置为声音缓冲区的开始处。然后，使用 Play 函数播放缓冲区中的音乐。该函数声明如下：

```
HRESULT Play( DWORD dwReserved1, DWORD dwReserved2, DWORD dwFlags );
```

示例程序如下：

```
lpDSBuffer->SetCurrentPosition( 0 );
lpDSBuffer->Play( 0, 0, 0 );
```

责任编辑　辛再甫　马征宇

软件高级编程实例精解丛书

C++ Builder 4 高级编程实例精解
Delphi 5 高级编程实例精解
Visual C++ 高级编程实例精解
Visual Basic 高级编程实例精解
C++ Builder 5 高级编程实例精解

丛书特色

软件开发的书籍，铺天盖地。但多病于——

- 或者有许多教条的知识点，点缀一些没有实用价值的小实例；
- 或者有许多的实例，但只能说是“示例”，无法让人深入其中；
- 或者高级宝典，纯粹“泊来”，那种“娓娓道来”总让真正高级编程者无可奈何！

本丛书有志于另辟新途，它——

- 只为有志于高级开发的人员量身定做，只有基本了解了该开发语言，才可能从该丛书中迅速而有效地“获益匪浅”；
- 按专题划分章节，每个专题贯穿一个实例，你不必拘泥于章节的先后顺序（除非特殊说明），直接寻找你正感兴趣或有疑惑的专题，深入研究即可；
- 每一个实例都是长期进行软件开发的作者心智的凝聚，而这些实例在大多数书籍中往往“只说不做”，因为它们每一个都会耗费开发者和作者相当大的心血；
- 配套光盘包含书中实例的所有源代码，可供借鉴使用，“含金量”非常高。

希望这套丛书，伴你在新的世纪一路凯歌，大步迈进“高级开发者”行列。



ISBN 7-118-02490-2/TP·580
定价:55.00 元